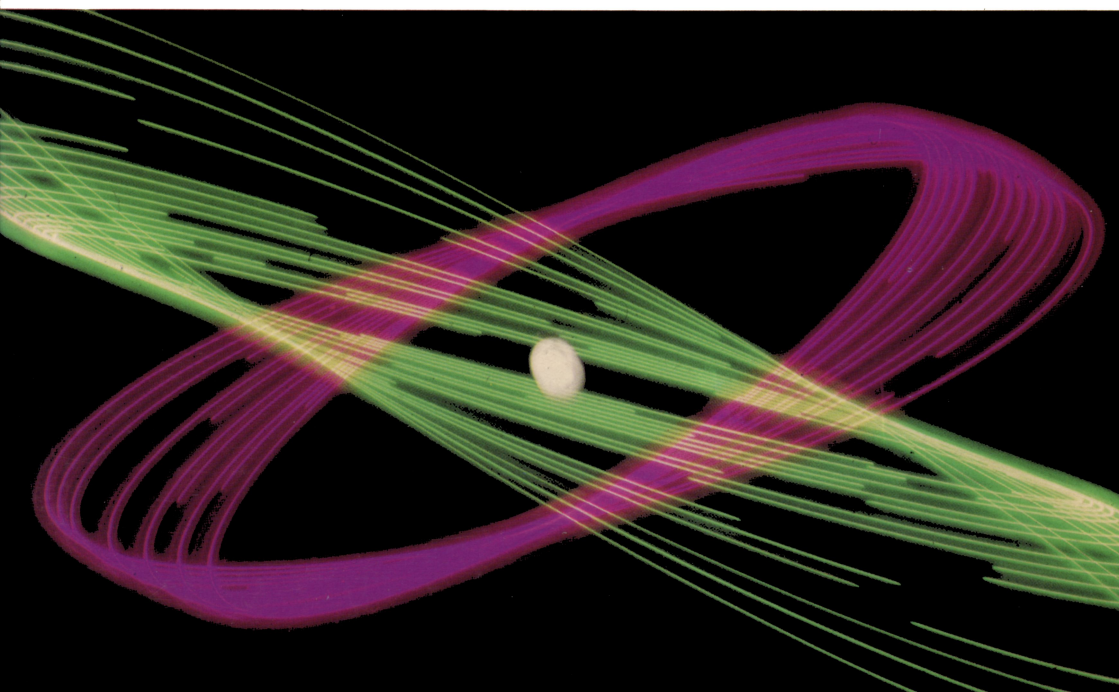
IMPARIAMO IL PASCAL

EDIZIONE ITALIANA

FLAVIO WALDNER

GRUPPO EDITORIALE JACKSON

IMPARIAMO IL PASCAL

di
Flavio
Waldner

GRUPPO
EDITORIALE
JACKSON
Via Rosellini, 12
20124 Milano

SOMMARIO

CAPITOLO 0

DA NON TRASCURARE

"Ogni uomo ha sempre da imparare anche su argomenti a lui già noti ed anche da persone che questi argomenti conoscono meno di lui"

(Anonimo)

Normalmente avrei scritto "introduzione". Però, siccome so che nessuno legge mai le introduzioni, allora ho pensato bene di cambiargli nome, così sono sicuro che il Capitolo 0 lo leggeranno tutti, non foss'altro che per curiosità. Ed è importante che tu lo legga.

Parliamo anzitutto di questo libro, e cominciamo anzi a parlare del suo argomento: il PASCAL. Si tratta di un linguaggi nuovo (anche se non tanto: il Prof. Niklaus Wirth lo ha scritto nel '72, all'Università di Zurigo) che sta conquistando vaste simpatie in molti ambienti e che forse è il miglior candidato alla successione di quel dominatore incontrastato della scena dei linguaggi che è sul trono da ben 22 anni: il FORTRAN. Anche se i FORTRAN di adesso hanno ben poco da spartire coi balbettii dei primi anni 60, essendosi grandemente evoluti e perfezionati, è da riconoscere sia che il linguaggio originale conteneva (e contiene) doti notevoli, dato che è riuscito a sopravvivere alle enormi novità tecnologiche nel frattempo avvenute, sia che 22 anni sono lunghi per tutti, e si sta attendendo qualcosa di meglio di un linguaggio nato ai tempi delle Schede e dei Nastri, delle memorie a ferrite (quindi piccole e costosissime) e dei calcolatori lenti e capaci di seguire un solo lavoro per volta.

Tentativi di altri linguaggi ce ne sono stati e parecchi di questi sono ancora sulla breccia (ti posso citare il PL/1, l'APL, il COBOL — orribile! —, l'ALGOL). Ma nessuno pare abbia intenzione di diffondersi altro che in ambienti particolari: il COBOL nel cosiddetto "calcolo finanziario", l'APL in casa IBM come linguaggio interattivo per terminali, l'ALGOL dormicchia dai tempi del FORTRAN di cui è contemporaneo, il PL/1 è nato vivacchia e morirà in casa IBM. Un'eccezione va fatta per il BASIC, spiccio e simpatico, anche se terribilmente limitato proprio per concezione e struttura.

In generale chi finora ha deciso della fortuna di un linguaggio è stata — incontestabilmente — la comunità scientifica. La quale oggi è però arroccata al solo FORTRAN che, potenziato da incredibili librerie matematiche, vere biblioteche di programmi di tutta la matematica creata dall'Uomo, è attualmente il dominatore incontrastato

della scena. Questa situazione però mostra parecchie crepe: con l'aumentare delle cognizioni è aumentata la complessità dei problemi da trattare, e non è raro il caso in cui ci si trovi a dover risolvere dei problemi per i quali la struttura stessa del FORTRAN risulta inadeguata. I problemi si possono risolvere lo stesso, intendiamoci bene, e di fatto si risolvono, ma il tutto lo si paga con programmi di complicazione e difficoltà di lettura estreme. E' abbastanza frequente ormai essere costretti a confessare che un programma scritto da un collega "non si capisce che diavolo faccia", ed occorrono settimane di opera di decrittazione per impadronirsi della situazione. Ed il guaio è che spesso la stessa cosa succede anche per programmi scritti da noi stessi, e ripresi magari a distanza di un paio di mesi.

Disordine? Disorganizzazione? Superficialità nella documentazione parallela (commenti, descrizioni, paginette dattiloscritte in cui si spiega ciò che il programma fa)? Certamente sì. Ma sarebbe troppo semplicistico attribuire queste difficoltà solo a cause di questo genere: in realtà **è il linguaggio stesso** che genera queste difficoltà. Nè del resto la storia umana è nuova a fatti di questo tipo: l'enorme progresso della matematica moderna è in gran parte dovuto all'introduzione di un simbolo speciale per i numeri (la "cifra") al di là degli unici simboli conosciuti prima: le lettere dell'alfabeto. Situazione che ha prodotto una conseguenza importante: quella di avere l'idea di usare le **lettere** per indicare dei **numeri qualsiasi**, aprendo le porte all'algebra moderna. Anche questa in fondo era all'inizio una semplice questione di linguaggio.

PASCAL si pone ad un livello decisamente superiore al FORTRAN per il calcolo scientifico: del FORTRAN ha tutte le possibilità, ma queste aggiunge una compattezza ed una concisione ed una chiarezza che all'altro sono sconosciute. Se e quando verrà accettato dalla comunità scientifica è da vedere e da dimostrare. Spero presto. Ma PASCAL ha anche un altro vantaggio: è un linguaggio molto adatto ai computer da tavolo la cui diffusione è da prevedersi assolutamente esplosiva, così come esplosiva è da prevedersi la loro evoluzione, in tempi in cui si parla di memoria di massa a bolle magnetiche (destinate a sostituire e a far relegare nel museo i dischi: forse i nastri no, data l'estrema loro economicità e praticità), di chips di CPU a 16 bits (e chissà che non spuntino i 32 bits?) e di nuovi semiconduttori all'arseniuro di gallio che farebbero saltare la velocità degli attuali calcolatori di un fattore $10 \div 100$. Il che vuol dire una cosa: i linguaggi del futuro **non** saranno più decisi dalla sola comunità scientifica, anche se questa avrà un rilevante peso nella loro scelta. E PASCAL appunto ha il notevole pregio di **non** essere solamente un linguaggio scientifico. E' adattissimo a calcoli gestionali ed a tutta quella parte della cultura scientifica stessa che non parla solo in termini strettamente quantitativi, ma è anche obbligata a parlare in termini — per così dire — descrittivi. Biologia, geologia, oceanografia, . . . spesso usano all'inizio di un'indagine degli apprezzamenti soggettivi, che poi via via diverranno quantitativi. E che dire delle scienze umanistiche, nelle quali il calcolatore è entrato ancora così poco? A tutte queste disparate situazioni — a parer mio — PASCAL è in grado di offrire uno strumento di possibilità convincenti.

E poi non dimentichiamo la massa dei commercianti e dei tecnici che per la prima volta si trovano in grado di offrirsi un calcolatore e che potranno gestire archivi altrimenti impossibili da controllare, o scroprire che un cervello umano con un calcolatore (ed un po' di mestieraccio!) possono fare meraviglie. A tutti costoro PASCAL è in grado di rispondere — a parer mio — con rara efficacia.

Aggiungi ancora una cosa: se è vero ciò che vedo succedere nel mondo scientifico (sensibilissimo barometro delle tendenze future), l'informatica degli anni 80 non produrrà chili e chili di carta stampata (di cui poi si legge sì e no il 5%) ma grafici, immagini, disegni. E per di più a colori. Oltre agli anni 80 non azzardo previsioni. Ma sono certissimo che l'uscita del calcolatore sarà sempre più un grafico, o un'immagine, possibilmente a colori (ah, le fotocopiatrici che sono ancora al bianco e nero!) e sempre meno il pacco della carta stampante "a soffietto", con lettura facilitata o non. Le stampanti tenderanno a diventare sempre più piccole (la carta costa . . .) e sempre più veloci, e produrranno pochi foglietti illustrativi del prodotto principale del calcolatore, che sarà **grafico**. Ebbene: anche in questo caso PASCAL è ben piazzato: un eccellente lavoro fatto dal MIT ha prodotto un "package" di routines grafiche estremamente veloci da usare. Il futuro naturalmente è ancora da scrivere.

Quindi — chiederai — se questa è la situazione, come mai PASCAL non si è ancora diffuso a macchia d'olio? Che si aspetta? Diciamolo pure: si aspetta che qualche grande Casa lo implementi nei suoi calcolatori (soprattutto la IBM). Però non basta. Occorre anche che ci sia una precisa richiesta da parte degli utenti. Cioè da parte tua. Senza questa i tempi si allungheranno.

E qui finalmente vengo allo scopo di questo lavoro che esce dopo una serie di articoli (sempre sul PASCAL) apparsi su BIT, e che mi hanno permesso anche di imparare il mestiere di porgere questi argomenti al "grosso pubblico" e non semplicemente ad un'auditorio di un'aula universitaria. Il fatto è che non c'è finora un testo che insegni a programmare in PASCAL. I libri attualmente nel giro sono o troppo stringati (come il celebre "User's Manual and Report" di Wirth e Jensen), e scritti per gente che già ha un bel po' di pratica (almeno di altri linguaggi) sulle spalle, o troppo semplici, anche se ben fatti. Il risultato è che il lettore medio pensa immediatamente a qualcosa che è fatta per aspiranti premi Nobel. E tutto finisce ancora prima di iniziare.

Questo libro vuole essere quindi un libro di divulgazione ed un vero e proprio corso di auto-istruzione, colmando una lacuna che se è vera in generale, è ancora più vera in Italia: infatti non va sottaciuto il fatto che tutti i libri su PASCAL sono in inglese. E questo per alcuni può non costituire un handicap, ma per molti può essere un'invalicabile barriera. Nè è detto che questi secondi siano poi sempre mentalmente inferiori ai primi!

Quindi una scelta di "didattica a largo raggio"? Forse; comunque ne derivano alcune conseguenze: uno stile anti-accademico per convinzione profonda e scelta ostinata. L'uso (provocatorio?) dell' "io" e del "tu", così desueto nei Sacri Testi, ma così simile allo "you" inglese, al tu dei latini, ed alle tradizioni della letteratura scientifica italiana. Un modo di parlare fatto apposta — insomma — per evitare in ogni modo che tu ti senta sui banchi di una scuola. So anche troppo bene che in Italia è una situazione psicologica da evitare accuratamente.

Altra conseguenza è la divisione della materia in capitoli il più possibile organici, in modo che rivederli sia semplice ed agevole. E con un riassunto del capitolo che è messo all'inizio e non in fondo proprio allo scopo che tu stesso possa controllare come procede passo passo il tuo lavoro. E — se vuoi — c'è sotto anche un sottile stimolo psicologico . . .

Poi ci sono dei consigli, su come usare questo libro: perchè è un libro che va

usato, non **letto**. Ancora una volta sono costretto a lanciare strali contro certa scuola, che obbliga i ragazzi a "studiare" per "essere interrogati". Il che vuol dire "imparare a memoria" per poi vedere se si è in grado di "recitare". No: questo non è il punto. La cultura scientifico-tecnica obbliga sì a studiare ed anche ad imparare a memoria (starebbe fresco un chimico che non si ricordasse i simboli degli elementi!) ma lo scopo non è di recitare, ma di **fare**. E qui è il punto. Non servirebbe a niente se tu sapessi le profonde differenze linguistiche e semantiche che esistono fra la DO del FORTRAN e la FOR di PASCAL, e poi non sapessi come usare la FOR, e come inserirla in un programma concreto. Sarebbe tempo sprecato, o forse peggio. Gli Americani hanno un'eccellente modo di dire per indicare coloro che non sono capaci di fare una cosa, ma conoscono tutti i libri e gli articoli di giornale e le notizie e notiziole sull'argomento: possono intrattenerti per giornate intere. Li chiamiamo i "paper experts": gli "esperti di carta". Non scendo nei particolari, perchè credo che troppi esperti di carta ci siano in giro; penso che la morale la possa tirare da solo.

Questo libro è fatto perchè tu lo usi per gradi: ti sconsiglio di fare più di un capitolo per giorno, se non hai precedenti esperienze di programmazione. C'è un tempo di assimilazione naturale che è bene non forzare troppo, pena la confusione. Comunque è chiaro che questo è solo un consiglio e che le persone si devono poi regolare da sole. Sempre però rimanendo nell'ordine dei consigli, sarebbe bene che tu potessi seguire un capitolo al giorno **ogni giorno**: questo ti porterà a chiudere il libro alla fine di due settimane circa, con PASCAL che non dovrebbe avere per te più alcun mistero.

Una parola sui problemi: ho cercato di metterne più che potevo, ma anche la mia inventiva ha dei limiti. Ce n'è comunque una buona scorta e dovresti farne il più possibile, al limite tutti, anche se non è tassativo il fatto che tu — avendo oggi letto il capitolo III — debba fare tutti i problemi del capitolo III, proprio oggi. Una certa ragionevole elasticità evidentemente è affidata al buon senso di chi legge. Però dovresti mantenere un passo di marcia tale che alla fine dell'ultimo capitolo tu abbia risolto una buona parte dei problemi. Diciamo un buon 75%.

Vedrai che ci sono dei problemi che ti dicono di spiegare con parole tue un argomento. Vedi di essere molto preciso nelle tue spiegazioni, come se dovessi rivolgerti a chi dell'argomento non ha la più pallida idea, e vedi se è possibile mettere le spiegazioni per iscritto. Ti sembrerà infantile, le prime volte: ma non lo sarà tanto se le spiegazioni tenterai di darle (o dartele) **senza** aprire il libro.

Quindi alla fine dovresti conoscere piuttosto bene PASCAL dopo circa due settimane. Non penso sia molto e penso che non troverai difficoltà insormontabili.

A questo punto non mi resta che augurarti buon lavoro e ricordarti che sono sempre a dua disposizione per suggerimenti e/o critiche che tu possa avere su questo testo. Nello spirito del motto iniziale.

F. WALDNER

Bari, 1981

CAPITOLO 1

COME SI DESCRIVE LA SINTASSI DEL LINGUAGGIO

SCOPO DEL CAPITOLO

Alla fine di questo capitolo sarai in grado di

. . . sapere cosa è un metalinguaggio

. . . sapere cosa sono gli elementi di un metalinguaggio

. . . sapere come si combinano fra di loro gli elementi di un metalinguaggio

. . . costruire delle metaformule per descrivere la sintassi di linguaggi particolari

. . . interpretare il formalismo di Backus-Naur (BNF per brevità)

. . . usare in pratica il BNF ed interpretare una metaformula scritta in BNF.

Questo capitolo non parla ancora di PASCAL: spiega invece solo il modo in cui PASCAL viene descritto in questo ed in altri testi. E' opportuno che tu sia molto disinvolto nell'uso del BNF, in modo da non incontrare difficoltà nei capitoli successivi: ti consiglio pertanto di **non** cominciare lo studio di PASCAL fino a quando questo capitolo non ti è del tutto ovvio e — per così dire — ti viene naturale. Di solito bastano due giorni per padroneggiare completamente il BNF. In ogni caso ricordati di risolvere i problemi e di rispondere alle domande proposte. Se dovessi incontrare qualche difficoltà rivediti il capitolo, ripassati i problemi e — magari — inventane qualcuno a tua volta. Ciò vale anche nel caso in cui le difficoltà sorgessero quando sei già un po' avanti con lo studio di PASCAL. Anche se quest'ultimo caso è da considerarsi piuttosto improbabile.

IL BNF: INTRODUZIONE

Prima di parlare del PASCAL sarà necessario fare una premessa: il linguaggio è impostato in modo rigoroso, professionale e sintatticamente corretto; pertanto va trattato di conseguenza. Tutto ciò succede fin dagli inizi: e poichè parlare di un linguaggio non è in genere facile, è stato deciso universalmente di adottare per PASCAL il formalismo di Backus-Naur (BNF d'ora in avanti) già da tempo in uso presso gli specialisti.

Vediamo di che si tratta. Fu nel 59 circa che John W. Backus dell'IBM si pose

il problema di come fare a parlare di un linguaggio senza dover spendere mari di parole, spesso di difficile se non di contraddittoria interpretazione. Fu così che nel '60 prese forma un vero e proprio sistema di regole e formule (molto semplice in verità, come vedremo) con cui si può descrivere un linguaggio in modo compatto e inequivocabile. Detto all'inizio **B**ackus **N**ormal **F**ormalism venne poi ribatezzato in **B**ackus **N**aur **F**ormalism come riconoscimento degli importanti contributi che P. Naur dette al suo sviluppo e diffusione.

Il BNF è un linguaggio con cui **si parla di** altri linguaggi. Ciò non è una novità nella logica matematica: questo tipo di attrezzo esiste da tempo, e viene chiamato **metalinguaggio** (ti ricordo che in greco *μετα* vuol dire "dopo, oltre, al di là"). Per analogia le formule del metalinguaggio sono chiamate **metaformule** ed i simboli, **metasimboli.**

Potremo quindi dire che un metalinguaggio usa dei metasimboli organizzati in metaformule per parlare di un linguaggio comune.

Nota a margine: le metaformule si chiamano anche **metaespressioni.**

Le analogie con l'algebra sono piuttosto evidenti: in questa si organizzano dei **simboli** in **formule** per descrivere delle **relazioni**. Per esempio quando scrivi semplicemente

$$S = \pi R^2$$

in realtà metti in moto un complesso meccanismo che si basa sulle seguenti convenzioni:

I) con π si indica quel tale numero (3.141598 . . .) ottenuto in un certo modo

II) con R si indica un altro numero

III) con un 2 scritto in piccolo in alto a destra di R si indica l'operazione di elevamento al quadrato di R

IV) l'operazione di cui al punto III) deve essere eseguita per prima

V) poichè fra π ed R non c'è scritto nulla, si sottintende un'operazione di moltiplicazione

VI) il segno = sta ad indicare la frase "si ottiene lo stesso numero che"

Insomma, se dovessimo dirla a parole la formuletta dell'area del cerchio suonerebbe pressappoco così:

"se si misura l'area del cerchio si ottiene lo stesso numero che si otterrebbe se se ne elevasse al quadrato il raggio e si moltiplicasse poi il risultato per π, definito come il rapporto fra la lunghezza della circonferenza ed il diametro di un qualunque cerchio".

E' piuttosto evidente che la formula è ben più pratica e concisa della descrizione a parole. Incidentalmente: nota che le formule sono un'invenzione relativamente recente, e che praticamente fino al '700 si andò avanti con descrizioni a parole come quella di cui sopra, per di più in latino.

La concisione è anche un pregio del BNF. Cominciamo quindi a vedere i vari

metasimboli:

I) **le parentesi angolari:** (⟨ ⟩). Entro queste parentesi vanno messe le descrizioni (a parole) dell'elemento di linguaggio di cui stiamo parlando. Scriveremo dunque:

⟨programma⟩ ⟨frase⟩ ⟨costante⟩

ed intenderemo "un qualunque programma", "una qualunque frase", "una qualunque costante".
Insomma, in certo senso le parentesi angolari con ciò che contengono costituiscono l'elemento base del linguaggio. Un po' come le lettere ed i numeri in algebre. Sono — insomma — delle **metavariabili.**

II) **Gli operatori.** Come si lavora con le metavariabili? Le operazioni di base sono molto più semplici che non quelle dell'algebra. In effetti ce ne sono solo due, e sono la AND e la OR logiche. Le convenzioni sono che al posto della AND (che nel BNF acquista il significato di "seguito da") non si scrive nulla, mentre al posto della OR si scrive una barretta verticale (|).

III) **Il simbolo relazionale:** questo va letto "consiste di" o "è definito come" e si scrive : : =

E con questo finiscono in pratica le regole del BNF puro.

ESEMPI SUL BNF ED ESTENSIONI

Per vedere come funziona il BNF facciamo un esempio, e supponiamo di voler esprimere in BNF la seguente definizione di una subroutine di FORTRAN

"una subroutine consiste di un'intestazione, di statements dichiarativi e di istruzioni"

Ci troviamo subito di fronte ad un primo intoppo che ci obbliga ad allargare il metalinguaggio, dal momento che noi sappiamo che l'**intestazione** non ha una forma qualunque, ma anzi ha una sintassi ben definita. E' obbligatorio l'uso della parola SUBROUTINE seguita da un nome, il quale a sua volta **può** essere seguito da una coppia di parentesi che contengono uno o più nomi di variabili o costanti: se questi sono più di uno, devono venire separati da virgole. Insomma, ecco i vari casi "legali"

```
SUBROUTINE TIZIO
SUBROUTINE CAIO  (A)
SUBROUTINE BEPPE  (A, B, C)
SUBROUTINE CECCO (ADELE, Z56, MILAN)
```

Come faremo ad esprimere tutto ciò in BNF? Posto che abbiamo già definito ciò che intendiamo come argomento di una subroutine (matrice, costante, variabile . . .) avremo comunque la necessità di amplicare il linguaggio nei seguenti modi

I) ci occorre una convenzione qualunque per indicare delle parole o dei simboli che sono esclusivi ed obbligatori nel linguaggio di cui stiamo parlando. Sono le cosiddette "reserved words" esemplificate dalla parola SUBROUTINE nell'esempio precedente.

II) ci occorre un qualche trucco per poter dire che una certa metavariabile può non esserci (cioè non è necessario metterla) può esserci, ed essercene una sola, e può anche darsi che ce ne siano tante.

Cominciamo dal punto I). Ci sono due tendenze, nel BNF: la prima è quella di scrivere la parole obbligate ("reserved words") fra apici (in tal caso scriveremmo 'SUBROUTINE') e la seconda che obbietta che l'apice è di solito un carattere di uso frequente nei linguaggi e che quindi è meglio evitare di fare confusione; a questo scopo preferisce la sottolineatura (in tal caso scriveremmo SUBROUTINE). In questo testo useremo la seconda alternativa, che è da considerarsi molto più pratica della prima e molto più evidente per chi legge. Ci sarà un'unica eccezione: quando una parola riservata è costituita **da un solo carattere non** verrà sottolineata. Questo non è un capriccio, ma risponde a delle semplici esigenze di chiarezza tipografica.

Le parole riservare, il cui uso in un linguaggio ha in genere un solo scopo, e non può essere mutato secondo la volontà di chi usa il linguaggio medesimo, si comportano in certo modo come le costanti nell'algebra: non a caso vengono dette **"metacostanti"**.

Vediamo ora come possiamo usare ciò che abbiamo definito finora: ti darò prima la definizione a parole e poi la metaformula corrispondente.

"una costante intera è definita come un intero privo di segno oppure preceduto da uno dei due segni + o –"

Ecco il BNF

⟨costante intera⟩ : : = ⟨intero non segnato⟩
| + ⟨intero non segnato⟩
| - ⟨intero non segnato⟩

Ecco un'altra definizione

"una cifra numerica consiste di una cifra qualunque compresa fra 0 e 9 inclusi"

Ed ecco il BNF

⟨cifra⟩ : : = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Ancora:

"una lettera è definita come una lettera dell'alfabeto inglese dalla A alla Z incluse"

il cui BNF diviene

⟨lettera⟩ : : = A | B | C | D | E | F | G | H | J | K | L | M | N | O | P
| Q | R | S | T | U | V | W | X | Y | Z

ed una definizione più complicata, che ha bisogno di queste definizioni preliminari potrebbe essere

"un carattere alfanumerico è definito come una cifra numerica o come una lettera"

ed il BNF diverrebbe

⟨carattere alfanumerico⟩ : : = ⟨lettera⟩ | ⟨cifra⟩

Quanto detto risponde all'esigenza di avere nel linguaggio la possibilità di introdurre delle metacostanti ed anche la possibilità di notarle in modo chiaro.

Non così ovvio è l'altro punto: abbiamo bisogno di avere un formalismo che ci permetta di dire che una metavariabile può non esserci, oppure può essercene una sola, oppure può essercene più d'una.

Ad una richiesta simile i puristi hanno già nel BNF come l'abbiamo definito noi finora una risposta inequivocabile. Per esprimere una situazione di questo genere scrivono

⟨A⟩ : : = ⟨nulla⟩ | ⟨A⟩ ⟨B⟩

Ciò — detto a parole — si esprimerebbe nel seguente modo

I) l'elemento A può non esserci (cioè "essere costituito da nulla")

II) se l'elemento A c'è può essere costituito o da B o da un altro elemento A. E qui delle due l'una
   a) o questo secondo elemento A è "costituito da nulla", e quindi non c'è, ed in questo caso il primo elemento A risultato costituito **da un solo elemento B.**
   b) o questo secondo elemento A è costituito da un B. Ed in questo caso il primo elemento A risulta costituito da **due** B.

Ed il ragionamento si può ripetere: la relazione suddetta prende il nome di "relazione ricorrente", è bella ed elegante, ma ha un unico difetto. Non è evidente e costringe la gente a pensare un po' troppo. E siccome la gente che lavora col calcolatore è molto pigra (per fortuna: il progresso è figlio della pigrizia!) sono stati inventati altri formalismi più semplici ed immediati.

Ecco quindi come si può scrivere il fatto che A può non esserci, essere costituito da un solo B o da una serie di B

⟨A⟩ : : = [<B>]
⟨A⟩ : : = {⟨B⟩}

Noi useremo la seconda alternativa, dal momento che la prima può generare qualche confusione (ci sono dei testi in cui però la potrai trovare).

Armati di questo nuovo strumento, vediamo come possiamo tradurre in BNF la definizione

"un numero intero è costituito da una sequenza di cifre, di cui almeno una deve essere presente"

⟨numero intero⟩ : : = ⟨cifra⟩ {⟨cifra⟩}

Così la definizione di una parola

"una parola è costituita da una sequela di lettere, ed occorre che sia presente almeno una lettera"

diviene in BNF

⟨parola⟩ : : = ⟨lettera⟩ {⟨lettera⟩}

Più complicata la definizione (frequente nei linguaggi di programmazione) di costante alfanumerica

"una costante alfanumerica consiste di una sequela di lettere e/o cifre, delle quali la prima (sempre necessariamente presente) deve essere una lettera"

Esempi possono essere T, A4, G675UH, N7UH9GT4, GHJKL mentre 3.56 non può essere riconosciuto come costante alfanumerica sia perchè comincia con un numero e non con una lettera e poi perchè contiene un . che non è nè una lettera nè un numero. Ecco il BNF

⟨costante alfanumerica⟩ : : = ⟨lettera⟩ {⟨cifra⟩ | ⟨lettera⟩}

Il vantaggio delle parentesi al posto della formula iterativa è anche un altro: è invalso l'uso di attaccare degli indici alle parentesi, in questo modo

$$\{B\}_3^9$$

che si legge: "B può essere ripetuto da un minimo di 3 ad un massimo di 9 volte".

Con questa ulteriore convenzione ecco cosa diviene il BNF della definizione di un identificatore in FORTRAN, che esplicitamente suona così

⟨identificatore FORTRAN⟩ : : = ⟨lettera⟩ $\{$⟨lettera⟩ | ⟨cifra⟩$\}_0^5$

Riassumiamo quindi le convenzioni del BNF in una tabella

⟨ ⟩ : parentesi angolari. Contengono la definizione di un elemento del inguaggio (metavariabile)

| : serve ad unire due metavariabili con una OR logica. La AND logica (da leggersi "seguito da") non si scrive: si scrivono semplicemente le due metavariabili una di seguito all'altra.

____ : sottolineatura. Serve ad indicare le parole riservate del metalinguaggio (reserved words o metacostanti).

: : = : connessione. Sta ad indicare la frase "è definito come".

$\{\}_m^n$: parentesi di ripetizione: indicano che gli elementi racchiusi all'intero delle parentesi possono essere ripetuti più volte: da un minimo di m ad un massimo di n volte.

A questo punto possiamo riprendere l'esempio iniziale della SUBROUTINE in FORTRAN e potremo scrivere

⟨identificatore⟩ : : = ⟨lettera⟩ $\{$⟨lettera⟩ | ⟨cifra⟩$\}_0^5$

⟨intestazione di SUBROUTINE⟩ : : = SUBROUTINE ⟨identificatore⟩

| SUBROUTINE ⟨identificatore⟩ (⟨identificatore⟩ {,⟨identificatore⟩})

Ti consiglio di prendere questa definizione in BNF e confrontarla con gli esempi che abbiamo fatto all'inizio di questo paragrafo. Potrai vedere da te che la definizione è estremamente sintetica e contiene tutta l'informazione necessaria a scrivere l'intestazione in modo sintatticamente corretto.

PROBLEMI ED ESEMPI

PROBLEMA I: vuoi costruire un catalogo per una biblioteca. Le informazioni che devi mettere sulla scheda di ogni libro sono le seguenti:

1) nome dell'autore
2) cognome dell'autore — (15 lettere al massimo a testa)
3) titolo del libro — (50 lettere al massimo)
4) luogo e data dell'edizione — (15 lettere al massimo per il luogo)
5) sigla del catalogo — (due gruppi di 6 cifre)

Possiamo esprimere queste richieste col BNF?

SOLUZIONE: supporremo di aver già definito cosa intendiamo per lettere e cifre numeriche. Potremo quindi scrivere:

⟨scheda bibliografica⟩ : : = ⟨nome⟩ ⟨cognome⟩ ⟨titolo⟩ ⟨luogo⟩ ⟨data⟩ ⟨sigla⟩

⟨nome⟩ : : = $\{lettera\}_1^{15}$

⟨cognome⟩ : : = $\{lettera\}_1^{15}$

⟨titolo⟩ : : = $\{lettera\}_1^{15}$

⟨data⟩ : : = $\{cifra\}_1^{9}$

⟨sigla⟩ : : = $\{cifra\}_1^{6}$ $\{cifra\}_1^{6}$

PROBLEMA II: prova a descrivere col BNF il tuo codice fiscale.

PROBLEMA III: definisci col BNF un numero telefonico completo di prefisso per teleselezione. Attenzione. Mentre il numero telefonico è una semplice sequenza di cifre, il prefisso obbedisce a regole precise: quali? Come ne tieni conto nel BNF? Saresti capace di scrivere il BNF per un numero telefonico internazionale? (attento qui al nuovo gioco dei prefissi).

PROBLEMA IV: descrivere col BNF le regole per costruire un polinomio.

SOLUZIONE: definiamo anzitutto due metavariabili in modo tale da poter parlare in modo conciso di quanto verrà dopo: definiremo come "somma algebrica" (abbreviazione S.A.) un'operazione che può essere a scelta una somma o una sottrazione. Così con l'abbreviazione P.A. intenderemo un'operazione che può essere una moltiplicazione oppure una divisione oppure un'elevazione a potenza.

Tutto quanto detto sopra può venir espresso in BNF nel seguente modo

⟨S.A.⟩ : : = ⟨somma⟩ | ⟨sottrazione⟩

⟨P.A.⟩ : : = ⟨prodotto⟩ | ⟨quoziente⟩ | ⟨elevamento a potenza⟩

Ti ricorderò a questo punto che un monomio è un insieme di un numero reale e di varie lettere legati fra di loro o da moltiplicazioni o da divisioni o da elevamenti a potenza. Inoltre un polinomio è una somma algebrica di uno o più monomi.

Questo complesso di definizioni può venir convenientemente descritto in modo

molto sintetico usando il BNF. Ecco come

⟨monomio⟩ : : = ⟨numero reale⟩ {⟨P.A.⟩ ⟨lettera⟩}

⟨polinomio⟩ : : = ⟨monomio⟩ {⟨S.A.⟩ ⟨monomio⟩}

E qui già puoi vedere che il BNF permetta di dare di un linguaggio una descrizione semplice, concisa e senza equivoci: è per questo che è stato deciso di usarlo per descrivere PASCAL: non è che usando il BNF si impari un linguaggio più in fretta dal momento che l'unico modo di imparare queste cose è infatti quello di usarle. Semplicemente la descrizione delle regole di sintassi viene molto semplificata, di facile consultazione e senza possibilità di fraintendimenti.

V) Spiega con parole tue in cosa, consiste un metalinguaggio, e qual'è la differenza fra linguaggio e metalinguaggio.

VI) Descrivi gli elementi fondamentali del BNF

VII) Scrivi il BNF di una targa automobilistica. Tieni presente anche il fatto che se il numero di targa passa il milione vengono introdotte delle lettere al posto dei numeri.

VIII) Descrivi con parole tue le convenzioni ed i simboli del BNF.

IX) Prova a scrivere il BNF dell'indirizzo di una lettera che debba essere spedita in Italia (all'Estero ci possono essere imprevedibili situazioni di descrizione dell'indirizzo!).

CAPITOLO 2

COME SI SCRIVE IN PASCAL

SCOPO DEL CAPITOLO

Alla fine di questo capitolo sarai in grado di

. . . sapere quali sono i caratteri ed i simboli ammessi per scrivere un programma in PASCAL

. . . conoscere le parole riservate (metacostanti), che in nessun caso possono essere usate per scopi diversi da quelli descritti nella sintassi

. . . sapere cosa si intende per **identificazione** e scrivere degli identificatori validi in PASCAL

. . . scrivere dei commenti

. . . sapere cosa si intende per **separatore** e cosa è un linguaggio a **formato libero**

. . . avere un'idea pratica di come si scrive una frase in PASCAL

COME SI SCRIVONO I NOMI IN PASCAL

Vediamo ora come si scrivono i nomi delle quantità che appaiono in PASCAL.

Ad ogni quantità che appare in un programma (sia essa una variabile, un sotto-programma, una matrice o altro) viene attribuito un **nome** che viene detto **identificatore**. Per costruire un identificatore legale occorre seguire alcune regole e si hanno a disposizione dei simboli. Altri simboli sono a disposizione per scopi diversi.

In sostanza il vocabolario di PASCAL consiste quindi in una **serie di simboli** e di **regole** per costruire gli identificatori (da notare che lievi differenze possono esistere fra vari sistemi). Oltre a ciò occorre ricordare che esiste una serie di **parole riservate** (reserved words o **metacostanti**) che in nessun caso possono essere usate fuori dallo uso che ne è previsto dal linguaggio stesso. Ecco il BNF di simboli e metacostanti.

⟨cifre⟩ : : = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

⟨lettere⟩ . : = A | B | C | D | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

⟨simboli⟩ : : = + | – | * | / | : = | . | , | ; | : | ' | = | ⟨⟩ | ⟨ | ⟨ = | ⟩ = | ⟩ | (|) | [|] | { | } | ↑ | "

⟨metacostanti⟩ : : = AND | ARRAY | BEGIN | CASE | CONST | DIV | DO | DOWNTO | ELSE | END | FILE | FOR | FUNCTION | GOTO | IF | IN | LABEL | MOD | NIL | OF | OR | PACKED | PROCEDURE | PROGRAM | RECORD | REPEAT | SET | THEN | TO | TYPE | UNTIL | VAR | WHILE | WITH

Alcune note:

I) non sono state sottolineate le metacostanti, ritenendolo implicito dal contesto, ed inutilmente complicato.

II) ti ricordo ancora una volta che è **assolutamente vietato** in PASCAL usare le metacostanti per scopi diversi da quelli previsti dal linguaggio stesso: per esempio non si può chiamare una variabile SET o DIV. Un buon metodo per evitare pene future è di spendere un po' di tempo ogni giorno, e di rileggersi ogni volta le metacostanti: dopo un po' si imparano a memoria e non si corre il rischio di dimenticarsele.

III) soprattutto per coloro che usano il FORTRAN: nota che nella GOTO non c'è uno spazio in mezzo. Attenzione: scrivere GO TO è una frequente causa di errore.

IV) vedrai che fra i simboli del linguaggio compare anche la parentesi graffa, che viene anche usata dal BNF per indicare ripetizione. Per non ingenerare confusione la parentesi graffa in PASCAL si usa **solo** per contenere i commenti, in modo da non generare complicazioni col BNF. In alcuni sistemi la graffa è sostituita dalla combinazione [* o dalla combinazione (*. Pertanto posso già anticiparti che tutti i commenti in PASCAL si possono scrivere in uno di questi due modi

{QUESTO E' UN COMMENTO}

oppure

[* QUESTO E' UN COMMENTO *]

oppure

(* QUESTO E' UN COMMENTO *)

Una novità rispetto ad altri linguaggi è il fatto che un commento può essere inserito dovunque in un programma, senza alcuna limitazione.

A questo punto possiamo vedere come si costruiscono gli identificatori cioè i nomi che nel programma vengono dati a qualcosa: sia questo qualcosa una variabile, un sottoprogramma, una matrice, una funzione od altro.

La struttura degli identificatori in PASCAL è uguale per tutti:

⟨identificatore⟩ : : = ⟨lettera⟩ {⟨cifra⟩ | ⟨lettera⟩}$_0^7$

con un'avvertenza: PASCAL accetta anche nomi più lunghi di quanto sopra specificato, solo che in ogni caso usa solamente i primi 8 elementi (siano essi cifre o lettere) per distinguere l'identificatore.

Così sono identificatori legali ABC ed anche PRECIPITEVOLISSIMEVOLMENTE, anche se in questo caso l'identificatore è sì accettato, però solo le prime 8 lettere sono usate per individuarlo (PRECIPIT). Pertanto se noi avessimo un secondo identificatore chiamato PRECIPITEVOLMENTE, questo **non** sarebbe capito da PASCAL come distinto dal primo: il calcolatore supporrebbe che stiamo parlando della stessa cosa (che chiamerebbe PRECIPIT). Invece un identificatore chiamato PRECIPIZIO sarebbe interpretato come un qualcosa di diverso (e chiamato PRECIPIZ). In generale è buona norma evitare di usare identificatori più lunghi di 8 lettere (sono molte, per la verità!) ad evitare possibili equivoci.

Nota che non sono ammessi simboli dentro gli identificatori, ma solo lettere o cifre, e che il primo elemento di un identificatore deve sempre essere una lettera.

IL CONCETTO DI SEPARATORE

Identificatori, metacostanti e simboli vari sono in certo senso le "parole" con cui si costruisce il linguaggio. E — come in tutti i linguaggi, — le parole vanno separate l'una dall'altra con delle convenzioni opportune. Nel linguaggio comune il carattere di separazione fra due parole è lo spazio, ma ci possono essere anche altre possibilità, quali per esempio i vari segni di interpunzione (combinati o non con spazi) ed altri simboli speciali quali le virgolette, le parentesi etc. Se provi a vedere quanti sono i caratteri di separazione (o le combinazioni permesse) nel linguaggio scritto, vedrai che la situazione è molto complessa a descriversi, anche se — evidentemente — noi siamo tanto abituati a leggere che non ci facciamo più caso.

In PASCAL la situazione è molto più semplice: i caratteri separatori sono solamente due: lo **spazio** ed il carattere di **"fine linea"**, qualunque esso sia nel particolare sistema in esame. Questa seconda caratteristica non dice molto finchè non ci rifletti un attimo: scrivere una frase in PASCAL, (come in qualunque linguaggio) significa scrivere di seguito certe parole separate fra loro in modo che il compilatore possa riconoscerle ad una ad una. Ebbene: la stessa funzione che entro una frase di PASCAL ha lo spazio la ha anche il carattere di "a capo". In altri termini: una frase di PASCAL continua **naturalmente** riga dopo riga, senza che ciò debba costituisce una preoccupazione per il programmatore. Ti anticipo inoltre che è possibile scrivere più di una frase (statement) su una sola riga. Queste due caratteristiche combinate danno come risultato il fatto che PASCAL è un linguaggio a **"formato libero"**, non ristretto come — p. es. — il FORTRAN ad avere **una frase per riga**, con eventuali righe di continuazione che devono in ogni caso venir esplicitamente dichiarate.

Altra novità è che **ogni commento è considerato alla stregua di un carattere separatore**. E ciò evidentemente permette a chi scrive di mettere commenti dove più gli aggrada, anche in mezzo ad una frase.

E' bene però subito dirti che sarà opportuno che tu non abusi di queste possibilità del linguaggio. Per esempio: è bene mantenere la convenzione "una frase per riga" almeno in linea generale, salvo magari mettere in una sola riga due o più frasi corte e magari in qualche modo connesse logicamente fra loro. Così pure è bene mettere i commenti **separati** dalle frasi di programma, salvo casi eccezionali. E tutto questo non perchè il linguaggio non lo permetta, ma perchè ci sono anche delle

esigenze di leggibilità del programma, e leggibilità di solito vuol dire ordine e pulizia: può darsi che scrivere un programma badando anche all'aspetto estetico della lista ti possa sembrare un po' eccessivo, ma sono sicuro che un tuo collega che dovesse prendere in mano il tuo programma magari per modificarlo apprezzerebbe molto l'ordine e la leggibilità.

I caratteri separatori in PASCAL possono essere inseriti **dovunque**, in ogni statement. Due sole eccezioni, anche queste piuttosto ovvie:

I) **non** si possono inserire caratteri separatori all'interno di metacostanti (scrivendo per esempio BOO LEAN invece di BOOLEAN)

II) **non** si possono inserire separatori all'interno di un identificatore (scrivendo ad esempio GIG ETTO al posto di GIGETTO)

Per il resto i separatori possono essere inseriti a piacere di chi scrive, uniche limitazioni restando sempre la leggibilità dello scritto e le preferenze personali.

RIASSUNTO DI QUESTO CAPITOLO

In questi riassunti che troverai alla fine di ogni capitolo verrà ripreso il BNF degli argomenti spiegati nel capitolo stesso. Alla fine del libro troverai il BNF dell'intero PASCAL, che ti potrà servire come utile riferimento in caso di dubbi sulla sintassi del linguaggio.

⟨cifra⟩ : : = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

⟨lettera⟩ : : = A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

⟨simbolo⟩ : : = + | – | * | / | : = | . | , | ; | : | ' | = | ⟨⟩ | ⟨ | ⟨= | ⟩= | ⟩ | (|) | [|] | { | } | ↑ | "

⟨metacostante⟩ : : = AND | ARRAY | BEGIN | CASE | CONST | DIV | DO | DOWNTO | ELSE | END | FILE | FOR | FUNCTION | GOTO | IF | IN | LABEL | MOD | NIL | OF | OR | PACKED | PROCEDURE | PROGRAM | RECORD | REPEAT | SET | THEN | TO | TYPE | UNTIL | VAR | WHILE | WITH

⟨identificatore⟩ : : = ⟨lettera⟩ {⟨lettera⟩ | ⟨cifra⟩ $_0^7$

⟨separatore⟩ : : = ⟨spazio⟩ | ⟨carattere di file linea⟩ | ⟨commento⟩

⟨commento⟩ : : = [* {⟨cifra⟩ | ⟨carattere⟩ | ⟨simbolo⟩} *]
| (* {⟨cifra⟩ ! ⟨carattere⟩ | ⟨simbolo⟩} *)

PROBLEMI ED ESEMPI

I) Decidi quali di questi identificatori è legale e quale non lo è. In quest'ultimo caso decidi anche il perchè gli identificatori in esame sono legali.

A ; A23 : SCF ; 23A ; RFT23ED ; ASC/DF; SET ; SETABC ;

SET ABC ; AND ; NADSET ; AND/SET ; AND.SET ; 3ABC
IDENTIFICATORE ; IDENTIFICATORELUNGO ; IDENTIFI-
CATORE LUNGO ; IDENTIFICATORELUNGHISSIMO ;

II) Ecco alcuni esempi di programma scritto in PASCAL: cerca di individuare gli errori in base a quanto hai appreso finora

```
PROCEDURE PRIMA (A, B, C : INTEGER);
VAR D: INTEGER;
BEGIN
   IF A 20 THEN
   BEGIN
      D: =1; IF B 10 THEN
      BEGIN
         IF C =5 THEN WRITELN ('B =', B)
   END ELSE WRITELN ('C =', C)
END;
WRITELN (1A, 2B, 3C);
END.
```

III) A parte questioni di leggibilità e di comodità personale, dì se è lecito scrivere IN PASCAL invece che:

```
WRITELN (A, B, C);
```

qualcosa come

```
WRITELN (A,
   B, C);
```

IV) con riferimento alla domanda precedente, dì se è lecito scrivere invece di

```
PROCEDURE SECONDA (X, Y, Z: REAL; N:INTEGER)
```

i seguenti statements

```
PROCEDURE SECONDA
   (X, Y, Z: REAL; N:INTEGER);

PROCEDURE SECONDA (X, Y, Z: REAL; N:
      INTEGER);

PROCEDURE  SECONDA                    (X, Y, Z: REAL; N: INTEGER);

PROCEDURE SECONDA (X, Y, Z: REAL; N: INTE
            GER),

PROCEDURE
   SECONDA
      (X, Y, Z: REAL,                       N: INTEGER);
```

V) Elenca almeno 10 metacostanti di PASCAL

VI) Si può inserire un commento all'interno di un'espressione, ed al posto di uno spazio?

VII) E' lecito scrivere

PROCEDURE SECONDA [*COMMENTO*] (X, Y, Z:REAL; N: INTEGER);

VIII) E' lecito inserire un commento all'interno di un identificatore? ed all'interno di una metacostante?

CAPITOLO 3

IL PROGRAMMA E LE DICHIARAZIONI IN GENERALE

SCOPO DEL CAPITOLO

Alla fine di questo capitolo sarai in grado di sapere

. . . cos'è un programma strutturato a blocchi

. . . cosa si intende per "blocco"

. . . perchè nella moderna tecnica di programmazione si usa la struttura a blocchi

. . . come si presenta in generale un programma scritto in PASCAL

. . . come è strutturato un blocco in PASCAL

. . . cosa si intende per "parte dichiarativa di un blocco"

LA STRUTTURA A BLOCCHI

Più che essere una tecnica di programmazione, questa è una vera e proprio filosofia del linguaggio usato nei programmi, ed è derivata da accurate indagini di logica simbolica, da cui risultava già molto tempo addietro essere questo il modo più efficiente per costruire un programma.

Se vuoi un po' di storia, ti dirò che già l'ALGOL (linguaggio ancora in uso, anche se con scarsa fortuna e diffusione) nel 1958 era un linguaggio strutturato a blocchi, così come il PASCAL che stiamo studiando. Nulla di nuovo, quindi. L'ALGOL ebbe la sfortuna di essere stato scritto in modo notevolmente farraginoso e tale da scoraggiare tutti coloro che avevano qualche intenzione di usarlo praticamente. Ed infatti fu il FORTRAN a conquistare il mondo tecnico e scientifico, anche se questo linguaggio nacque con pretese assai modeste; infatti lo denuncia il suo stesso nome: FORTRAN deriva da "**FOR**mula **TRAN**slator" cioè "traduttore di formule" — Nome molto umile: la fortuna di questo linguaggio in questi 22 anni di sua presenza sul mercato è certamente ben superiore a quella che si poteva all'inizio attribuire ad un "traduttore di formule".

La struttura a blocchi si spiega molto meglio con un paio di schizzi che non a parole. a questo punto ti rimando alla fig. 1. In questa puoi vedere due diversi schemi di uno stesso programma (chiamato A1, per semplicità). Il primo si dice "schema

blocchi" (block structure) ed il secondo "schema ad albero" (tree structure). Ti dirò subito che in genere io preferisco di gran lunga il secondo.

In questo programma A1 esistono dei "sottoprogrammi" chiamati A2, A3, A4, B1, B2 che vengono chiamati da A1 e che si chiamano fra di loro, secondo lo specchietto seguente:

A1 chiama A2 che chiama A3 ed A4
A1 chiama B1 che chiama B2

però — evidentemente — A3 **non** chiama A3, nè A2 chiama B2. Gli schemi di fig. 1 ti evidenziano chiaramente la situazione.

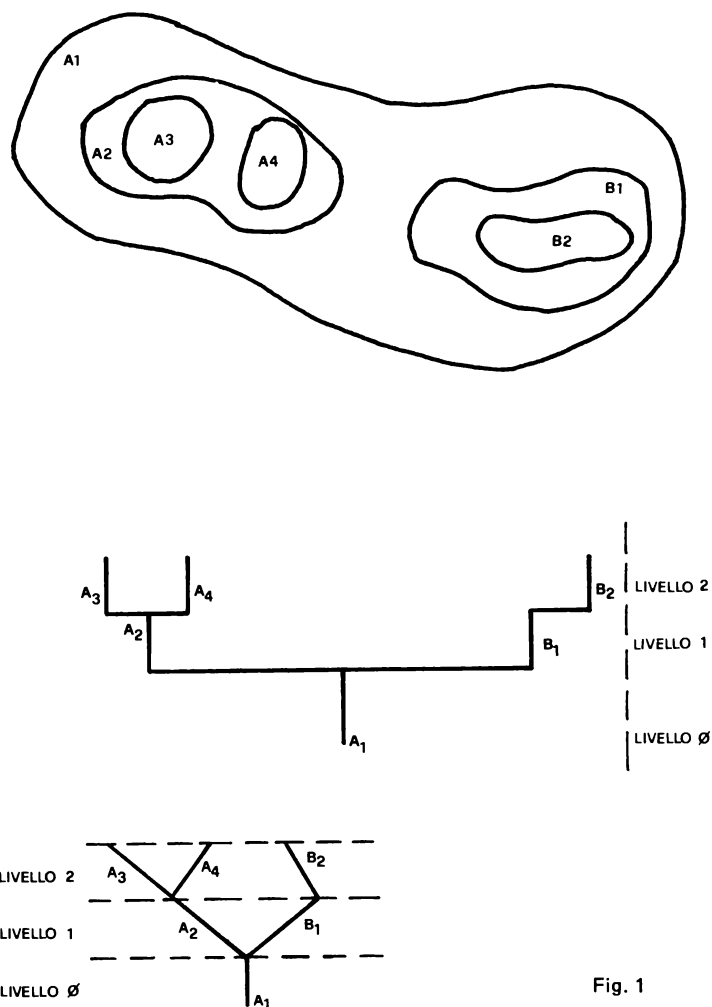



Fig. 1

Nella descrizione "ad albero" A1 viene detto appartenere al "livello zero" (è insomma il tronco dell'albero). A2 e B1 sono chiamati direttamente da A1 ed appartengono al primo livello (sono i rami principali, direttamente attaccati al tronco). A3, A4, B2 appartengono al livello 2: sono i rami secondari, attaccati ai rami principali.

Ebbene: in PASCAL **tutto** ciò che è definito in A1, di qualunque natura sia (variabili funzioni o altro) è accessibile a **tutti** i rami che ad A1 sono direttamente o indirettamente attaccati. Cioè a tutto l'albero. E' per questo che i simboli definiti in qualunque modo in A1 si dicono anche "globali".

Tutto ciò che viene definito in un ramo di un certo livello, è definito automaticamente nei livelli successivi, per tutti i rami che direttamente o indirettamente sono attaccati al ramo in questione. Per esempio: ciò è definito in A2 è "capito" anche da A3 e da A4, ma **non** da B2. E nemmeno da A1, dal momento che A1 si trova ad un livello inferiore (che corrisponde, per così dire, ad un superiore grado gerarchico) di A2. I simboli definiti in questo modo prendono il nome di simboli "locali".

Nella struttura ad albero la cosa è particolarmente ben evidenziata: la definizione dei simboli è molto simile alla linfa che scorre sempre verso i livelli superiori (e non al contrario!) e non può scorrere fra rami non collegati direttamente.

Per i fortranisti sarà interessante notare che in un linguaggio concepito in questo modo non c'è alcun bisogno del COMMON, essendo esso già implicito in quanto sono venuto dicendo. E se hai avuto già qualche esperienza di costruzione di un programma troppo grande per il calcolatore a disposizione, e hai dovuto ricorrere a tecniche di "overlay" (in cui solo un pezzo di programma può risiedere di volta in volta in memoria, mentre i risultati delle elaborazioni dei pezzi devono poter essere conosciuti da tutto il programma), la struttura ad albero non sarà certo per te una novità. Però sarà una novità — forse — impostare un programma che **fin dall'inizio** è scritto in un linguaggio strutturato ad albero: infatti, qualora ti dovesse capitare di avere un programma troppo grosso, che supera la capacità del tuo calcolatore, il ricorso alla tecnica di overlay sarebbe semplicissimo (per non dire banale) dal momento che il programma strutturato ad albero è già scritto nel modo adatto. In altri termini: in PASCAL tu scrivi fin dall'inizio il programma in modo tale da poter essere usato in overlay. E questo non è un vantaggio da poco.

Ti chiarisco per ultimo questo discorso della struttura a blocchi o ad albero in modo esplicito:

tutto ciò che è definito in	**è definito anche in**
A1	A1 A2 A3 A4 B1 B2
A2	A2 A3 A4
A3	A3
A4	A4
B1	B1 B2
B2	B2

Un "blocco" è quindi un programma, che può al suo interno contenere anche altri programmi, da lui direttamente chiamati. Tutto ciò che è definito nel blocco è

patrimonio del blocco stesso ed è accessibile, senza ulteriori difficoltà, da parte di tutti i programmi **che del blocco stesso fanno parte.**

La struttura a blocchi consente una visione molto chiara e razionale del problema, ogni blocco presentandosi come un'entità autosufficiente e dedicata alla risoluzione di una parte del problema stesso. Vantaggio non indifferente di questo tipo di struttura è la possibilità di passare da un programma interamente contenuto nella memoria del calcolatore, ad un programma che vi è contenuto "ad un pezzo per volta", tramite le tecniche di overlay. Dal momento che si diffondono sempre più i mini ed i micro, con capacità di memoria non eccessiva, capirai che la possibilità di usare un linguaggio che — ove se ne presenti la necessità — permette di passare alla struttura ad overlay senza alcuno sforzo costituisce un notevole vantaggio. In questo senso potremmo veramente dire che la struttura a blocchi di un linguaggio è più importante per i piccoli calcolatori di quanto non lo sia per i grossi mostri.

LA STRUTTURA DI UN PROGRAMMA

Cominciamo con una serie di definizioni e di terminologie.

Un **programma** (program), detto anche **algoritmo** (algorithm) si presenta in PASCAL come una serie di **azioni** (actions) che operano su dei **dati** (data). In PASCAL i dati vengono descritti da una serie di **dichiarazioni** (declarations) e di **definizioni** (definitions) mentre le azioni sono specificate da **istruzioni** (instructions).

La struttura generale di un **programma** è quella di un **blocco** (block) in cui sono contenute tutte le dichiarazioni, definizioni ed istruzioni, ed eventualmente anche i sottoprogrammi. Anche i sottoprogrammi sono dei blocchi a loro volta, secondo quanto descritto nel paragrafo precedente. E la struttura ad albero o a blocchi può continuare indefinitivamente: il fatto che un blocco sia racchiuso in un altro viene detto anche "nesting" con un termine intraducibile che deriva da "nest" (nido) e che vuol dire "stare racchiuso dentro, completamente, come in un nido". Come nelle scatole cinesi o nelle bambole russe.

La struttura del **blocco principale** (il programma, appunto) è quella di un blocco che contiene tutte le dichiarazioni, definizioni, istruzioni, sottoprogrammi, e che è preceduto da una **intestazione** (heading). Il BNF generale si presenta così:

⟨programma⟩ : : = ⟨intestazione⟩ ⟨blocco⟩

⟨intestazione⟩ : : = PROGRAM ⟨identificatore⟩
(⟨identificatore di file⟩
{,⟨identificatore di file⟩});

⟨identificatore di file⟩ : : = ⟨identificatore⟩

Vedremo più avanti cosa sono le files: per ora ti basti sapere che in generale almeno un identificatore di file deve essere sempre presente nell'intestazione del programma: questa file verrà usata per l'output del programma medesimo.

Eccoti — per esempio — un esempio di intestazione di un programma; per acqui-

stare maggiore sicurezza ti rimando al solito ai problemi alla fine del capitolo

PROGRAM PASTICCIO (INPUT, OUTPUT, FILAMIA, FILATUA, FILASUA)

In questa intestazione è da notare che l'identificatore è di 9 lettere (contro le 8 necessarie a PASCAL per distinguere un identificatore dall'altro) ed inoltre che le files INPUT ed OUTPUT (di ovvia funzione!) non necessariamente devono essere le prime due della lista, nè chiamarsi così. Le altre files con i nomi di fantasia sono delle files che in genere si suppone contengano dati da essere introdotti nel programma, o prodotti dal programma medesimo. Di queste files ce ne possono essere parecchie, a seconda della necessità del programma.

COME SI COSTRUISCE UN BLOCCO

Un blocco ha sempre la stessa struttura: ad una intestazione (heading) che può essere quella già vista, od altra che impareremo a conoscere più avanti, seguono alcune righe di dichiarazioni e definizioni. Quindi seguono le vere e proprie istruzioni del programma, racchiuse fra le parole BEGIN . . . END.

La parte delle dichiarazioni è estremamente importante, in quanto è bene ricordare che **tutti** i dati, variabili, matrici, insomma tutto ciò che in qualche modo viene usato in un programma PASCAL **deve** venire esplicitamente dichiarato, nei dettagli. Ciò a molti sembra un'inutile complicazione, ma non è affatto così: dichiarare tutto ciò che si usa, definendone le caratteristiche è un'eccellente pratica che evita molto spesso equivoci e mali di testa.

E' inoltre altrettanto eccellente pratica non mettere dichiarazioni in modo più o meno casuale, ma seguire degli ordini prefissati: per esempio l'ordine alfabetico. Ciò permette facili controlli consentendo: — al solito — di usare il tuo programma anche a chi non l'ha scritto.

Vediamo ora il BNF di un blocco: oltre all'intestazione (il titolo della commedia) c'e la lista degli attori (cioè le dichiarazioni e le definizioni di tutto ciò che nel blocco verrà usato) ed infine arriva il vero e proprio copione (cioè le istruzioni) che inizia con l'alzata del sipario tramite la parola riservata BEGIN e si chiude col "cala la tela"; che è la parola riservata END, seguita da un ; nel caso si tratti di un blocco generico (è il fine atto, insomma) e da un . nel caso si tratti della fine del programma, cioè dell'intera commedia.

La lista degli attori obbedisce a precise regole gerarchiche, e così succede anche in PASCAL. Ecco il BNF di un blocco:

```
⟨blocco⟩   : : =   ⟨dichiarazioni di LABELS⟩
                   ⟨definizioni di COSTANTI⟩
                   ⟨definizioni di TIPI⟩
                   ⟨dichiarazioni di VARIABILI⟩
                   ⟨dichiarazioni di PROCEDURE e FUNZIONI⟩
        BEGIN      ⟨istruzioni⟩ END.
```

E' molto importante notare che queste dichiarazioni e definizioni **devono** seguire l'ordine che ti ho specificato qui sopra, e **non è permesso** cambiare quest'ordine.

Siccome però non è facile ricordarsi l'ordine esatto delle dichiarazioni, sono state inventate alcune frasette che aiutano la memoria. Per esempio che PASCAL è

La **C**osa **T**re **V**olte **P**iù **F**acile

giusto per ricordarsi le iniziali di **L**abels, **C**onstants, **T**ypes, **V**ariables, **P**rocedures, **F**unctions. Puoi ovviamente inventarti altre frasette a trucco di tuo gradimento: l'importante è che ti ricordi l'esatto ordine in cui le dichiarazioni devono venir fatte.

RIASSUNTO DI QUESTO CAPITOLO

Un programma in PASCAL è strutturato a blocchi (o ad albero). In una struttura ad albero ciò che è definito ad un livello è definito automaticamente anche in tutti i livelli superiori ad esso collegati.

```
⟨programma⟩              : : = ⟨intestazione⟩ ⟨blocco⟩

⟨intestazione⟩           : : = PROGRAM ⟨identificatore⟩
                               (⟨identificatore di file⟩
                               {,⟨identificatore di file⟩}) ;

⟨identificatore di file⟩: : = ⟨identificatore⟩

⟨blocco⟩                 : : = ⟨dichiarazioni di LABELS⟩
                               ⟨definizioni di COSTANTI⟩
                               ⟨definizioni di TIPI⟩
                               ⟨dichiarazioni di VARIABILI⟩
                               ⟨dichiarazioni di PROCEDURE e FUNZIONI⟩
                    BEGIN      ⟨istruzioni⟩ END.
```

PROBLEMI ED ESEMPI

I) Controlla queste intestazioni di programmi e decidi quali sono errate e perchè:

```
I)     PROGRAM UNO (A, B);
II)    PROGRAM DUE (A, B, TIZIO, BEPPE, NANDO);
III)   PROGRAM TRE (INPUT, OUTPUT);
IV)    MAIN PROGRAM (INPUT, OUTPUT);
V)     PROGRAM DOWNTO (A1, A2, A3);
VI)    PROGRAM QUATTRO (SET1, SET2, FILE, FILE1);
VII)   PROGRAM CINQUE (A, B, C)
VIII)  PROGRAM
       SEI (FILA1, FILA2, FILA3, FILA4);
IX)    PROGRAM SETTE (G,
                H,L);
X)     PROGRAM OTTO (NOMELUNGO, NOMELUNGHISSIMO);
```

II) Si possono

I) dichiarare i tipi prima dei labels?

II) dichiarare le costanti dopo i tipi?
III) dichiarare i labels dopo le variabili?
IV) dichiarare le costanti dopo le procedure?
V) dichiarare le funzioni prima delle variabili?
VI) dichiarare i tipi dopo le variabili?
VII) dichiarare le funzioni prima dei tipi?
VIII) dichiarare le funzioni dopo le costanti?
IX) dichiarare le variabili dopo i labels?
X) dichiarare le variabili dopo le costanti?

III) Con riferimento alla figura di questo capitolo che descrive l'ossatura generale di un ipotetico programma supponiamo che XA1 sia una variabile definita nel blocco A1 ed YA2 sia una variabile definita nel blocco A2

I) XA1 è una variabile globale? e — se non lo è — perchè?
II) YA2 è una variabile globale? e — se non lo è — perchè?
III) se mi trovo nel blocco A3, posso usare senza preoccupazioni la variabile YAZ, mantenendole lo stesso significato e valore originario?
IV) se mi trovo nel blocco A4 e modifico il valore della variabile XA1, e se poi torno nel blocco A1 cosa trovo come valore di XA1: il valore che aveva prima di chiamare A4, o il nuovo valore modificato?
V) Supponiamo che YA2 abbia il valore 1, e che noi ci troviamo nel blocco A2. Ora il controllo si sposta nel blocco A3 nel quale ad YA2 viene assegnato il valore 5. Il controllo del programma torna successivamente al blocco A2 e da questo passa ad A4, senza che YA2 venga più modificata: che valore ha in questo momento?

IV) Nel corso dello sviluppo del programma ci si accorge che in A4 è stato definito un tipo speciale, che sarebbe molto utile avere definito anche in B1. Come si può fare per raggiungere questo scopo? La situazione sarebbe diversa se invece di trattarsi di un tipo si trattasse di una variabile che deve mantenere i suoi valori sia in A4 che in B2?

V) Solo per i fortranisti: è necessario qualcosa di simile al COMMON in PASCAL? potresti spiegare le ragioni della tua risposta (sia che essa sia positiva che negativa)?

VI) Una nota importante: sempre con riferimento alla figura di questo capitolo supponiamo che XXX sia una variabile definita nel blocco A2, e che anche nel blocco A4 ci sia ancora una definizione di XXX. Ebbene: in questo caso PASCAL (quando il controllo del programma è al blocco A4) accetta per buona **la definizione più recente.**
Sapresti indicare qualche uso di questa particolarità del linguaggio?

VII) Descrivi con parole tue in cosa consiste una struttura a blocchi o ad albero e che differenza passa fra due variabili con scopo locale e globale.

VIII) Se hai precedenti esperienze di programmazione: in cosa ti può aiutare la struttura a blocchi qualora tu volessi costruire un programma in "overlay"?

CAPITOLO 4

LE DICHIARAZIONI ED I TIPI STANDARD

SCOPO DEL CAPITOLO

Alla fine di questo capitolo sarai in grado di sapere

. . . cosa sono e come si definiscono i LABELS

. . . cosa si intende per COSTANTE, come la si scrive e come la si dichiara

. . . cosa si intende per TIPO

. . . cosa sono i tipi standard

. . . cos'è e come si opera sul tipo INTERO

. . . cos'è e come si opera sul tipo REALE

. . . cos'è e come si opera sul tipo BOOLEANO

. . . cos'è e come si opera sul tipo CARATTERE

. . . cosa sono le PROCEDURE e le FUNZIONI

. . . come si dichiarano le VARIABILI

. . . come si dichiarano i TIPI

. . . come si dichiarano le PROCEDURE

. . . come si dichiarano le FUNZIONI

I LABELS

I labels sono un residuo di linguaggi di programmazione più antichi, ed è bene che tu ricordi che meno ne farai uso meglio sarà in generale, sia per te che per la chiarezza del programma che scriverai. In generale PASCAL non ha affatto bisogno di labels, però può darsi che in qualche caso possano tornare utili: questa è l'unica ragione per cui non sono stati aboliti del tutto.

Cosa sono? Sono dei numeri da 1 a 4 cifre che vengono premessi da un'istruzione e che permettono successivamente di far riferimento all'istruzione medesima, magari deviando il flusso normale del programma e dicendo esplicitamente p.es. "salta all'istruzione 32". L'istruzione di salto la vedremo più avanti

e si chiama GOTO. In genere PASCAL **non** ha bisogno di istruzioni di salto di questo tipo: la sua logica è così potente e flessibile da rendere praticamente inutile questo tipo di istruzioni.

Possiamo quindi vedere il BNF sia dei labels che della dichiarazione dei labels

⟨label⟩ : : = {⟨cifra⟩}$_1^4$

⟨dichiarazione di label⟩ : : = LABEL ⟨label⟩ { ,⟨label⟩ } ;

e mentre al solito ti rimando agli esempi e problemi per vedere come in pratica si fa a dichiarare dei labels, eccoti un esempio tanto per chiarire il concetto

LABEL 374,55 , 7895 , 99;

In ogni caso sarà bene ripetere ancora una volta la raccomandazione di usare i labels se e solo se ciò è assolutamente inevitabile ed indispensabile. In generale meno si usano e più si è progrediti verso i concetti e le tecniche della programmazione strutturata, cui PASCAL naturalmente mira.

LE COSTANTI

Non sempre è necessario dichiarare le costanti che si usano nel programma: se un numero ad un certo punto deve venir moltiplicato per due, sarebbe molto grave che il "due" dovesse sempre essere esplicitamente dichiarato come un numero che verrà usato nel programma! In compenso spesso ci si trova ad usare delle costanti che magari sono il risultato di calcoli complicati: ed allora la loro definizione tramite un simbolo, e la loro dichiarazione esplicita (magari accompagnata da un commento) diviene di estrema utilità.

Vediamo anzitutto cosa sono le costanti. Esse possono essere di due tipi:

I) **un numero:** che viene scritto sempre in notazione decimale

II) **una stringa:** che viene definita come una serie di lettere o numeri o simboli, insomma di caratteri legali in PASCAL, i quali devono venir racchiusi fra **due apici**. Nel caso si voglia scrivere un apice questo deve venir ripetuto **due** volte di seguito.

Vediamo il BNF, che ti anticipo subito essere piuttosto complesso

⟨definizione di costanti⟩ : : = CONST ⟨identificatore⟩ = ⟨costante⟩;
{⟨identificatore⟩ = ⟨costante⟩ ; }

⟨costante⟩ : : = ⟨numero non segnato⟩ |⟨segno⟩ ⟨numero non segnato⟩
|⟨identificatore di costante⟩
|⟨segno⟩ ⟨identificatore di costante⟩
|⟨stringa⟩

⟨numero non segnato⟩ : : = ⟨intero non segnato⟩ | ⟨reale non segnato⟩

⟨intero non segnato⟩ : : = ⟨sequenza di cifre⟩

⟨sequenza di cifre⟩ : : = ⟨cifra⟩ {⟨cifra⟩}

⟨reale non segnato⟩ ::= ⟨intero non segnato⟩ . ⟨sequenza di cifre⟩
|⟨intero non segnato⟩ . ⟨sequenza di cifre⟩
E ⟨fattore di scala⟩
|⟨intero non segnato⟩ E ⟨fattore di scala⟩

⟨fattore di scala⟩ ::= {⟨cifra⟩}$^{2}_{1}$ | ⟨segno⟩ {⟨cifra⟩}$^{2}_{1}$

⟨segno⟩ ::= + | –

⟨identificatore di costante⟩ ::= ⟨identificatore⟩

⟨stringa⟩ ::= '⟨carattere⟩ |⟨simbolo⟩ {⟨carattere⟩ |⟨simbolo⟩} '

Non ci sono particolari commenti da fare: nonostante l'impressionante complicazione del BNF dovuta più alla necessità di completezza che non alla difficoltà vera e propria (in pratica le costanti in PASCAL si scrivono come negli altri linguaggi cui forse qualcuno di voi è già abituato), una sola è la novità di rilievo: in PASCAL si possono usare degli identificatori **con segno** anche quando si dichiarano le costanti. E' una comodità in più: per il resto non esistono novità e ti converrà vedere la parte pratica negli esempi e problemi.

I TIPI IN GENERALE

Al solito le definizioni generali non dicono molto: è infatti un po' difficile spiegare in astratto ed a parole cosa sia un "tipo". Potremo dire per spiegare il concetto che si possono effettuare operazioni solo fra oggetti "dello stesso tipo". Così, mentre esistono regole per effettuare operazioni tra variabili logiche o tra variabili reali, non si possono mescolare fra di loro questi due "tipi" di quantità.

In genere le regole da usare nel maneggio di tipi diversi sono più restrittive in PASCAL che non negli altri linguaggi, ed anche questa è una restrizione più apparente che reale, intesa non a rendere complicata la vita al programmatore, ma a rendere più semplice e più organizzato il suo lavoro, anche a costo di una disciplina più stretta.

Le variabili in PASCAL sono di due generi fondamentali: possono essere **strutturate** (matrici, records, files etc: tutte queste le vedremo più avanti) o **scalari**, cioè singole.

Le variabili scalari, di cui parleremo ora, possono essere **di qualunque tipo.** E questa è una grossa novità in PASCAL: è il programmatore stesso che può **inventare** nuovi tipi di variabili, quali il colore dei capelli di una persona, o il giorno della settimana, o le note musicali, ed avere a disposizione alcune operazioni per lavorare tra variabili di questi tipi di fantasia. Tornerò più avanti su questo argomento, ma ti posso anticipare fin d'ora che si tratta di un'enorme comodità del linguaggio, che permette un'incredibile agilità nella stesura di un programma, senza che sia necessario ricorrere a trucchi complicati per trattare tutti quei problemi (in genere gestionali) in cui compaiono variabili non numeriche.

Ogni sistema PASCAL **deve** però avere a sua disposizione quattro tipi **standard**: essi sono tipi intero, reale, booleano, carattere. Vediamone le caratteristiche.

IL TIPO INTERO (INTEGER)

Detto anche – evidentemente – "INTEGER" è costituito dai numeri interi, positivi o negativi, che certamente conoscerai.

Se però ti capiterà in mano un manuale di PASCAL troverai magari una frase del tipo "il subset degli integer consentito dall'implementazione del sistema". E credo che a questo punto resterai un po' perplesso, probabilmente. Niente paura: è un modo difficile per dire cose semplici; la strana frase infatti sta a significare che il tipo INTEGER non comprende **tutti** i numeri interi, ma solo una parte di essi (cioè un sottoinsieme, detto con un pessimo inglesismo "subset"). E ciò è evidente: se si ha a disposizione un calcolatore da 16 bits gli interi devono essere compresi fra –32768 e +32768. Al di fuori di questo **sottoinsieme** dei numeri interi non siamo più nel tipo INTEGER: più semplicemente viene data una segnalazione d'errore!

Fra entità di questo tipo si può operare con una serie di operazioni che elenchiamo qui di seguito, coi rispettivi simboli o parole (che sono parole riservate, cioè metacostanti!)

* moltiplicazione: il risultato della moltiplicazione di due numeri interi è ancora un numero intero

+ addizione: idem come sopra

– sottrazione: idem come sopra

In realtà dovrei essere più preciso: non necessariamente i risultati di queste operazioni danno dei numeri interi, visto che possono dare anche delle segnalazioni di errore. Se per esempio volessimo, in un calcolatore a 16 bits moltiplicare 10000 per 10000 otterremmo 100.000.000 che **non è** un INTEGER in quanto supera la massima capacità del calcolatore. In questo caso si ottiene infatti una segnalazione di OVERFLOW.

Le altre operazioni riguardano la divisione, che nel caso dei numeri interi deve essere indicata con simboli speciali (a differenza del FORTRAN che usa in ogni caso lo stesso simbolo "/"). Ecco quindi le operazioni

DIV : questa è l'operazione di divisione fra due interi, che in realtà è l'operazione di divisione **con troncamento.** Per spiegare più chiaro al solito è bene ricorrere ad un esempio:

7 DIV 2 dà come risultato 3 (intero)
12 DIV 5 dà come risultato 2 (intero)

e così via, naturalmente.

MOD : questa è un'operazione spesso utilissima, che in linguaggio difficile si scrive

A MOD B significa A – ((A DIV B) * B)

mentre in linguaggio semplice, cioè con un paio di esempi, diviene più comprensibile

25 MOD 10 dà come risultato 5
73 MOD 7 dà come risultato 3

l'operazione MOD dà quindi come risultato il **resto** della divisione.

Questo completa il panorama delle operazioni permesse nel tipo INTEGER. Se sei abituato a programmare in FORTRAN o in BASIC ti avverto di fare molta atten-

zione all'operazione DIV, che ti può causare qualche errore all'inizio, finchè non ti ci abituerai.

Ti faccio anche notare che nel PASCAL originale **non** esiste alcun simbolo per l'esponenziazione: Se vuoi fare l'operazione 2^5 sei nei guai. Questa restrizione, messa per motivi di non meglio identificate "buone abitudini di programmazione", è in realtà a dir poco discutibile, e praticamente tutti i PASCAL che vengono proposti dalle Case hanno ormai la doppia stellina (**) o l'accento circonflesso (^) come simbolo di esponenziazione. Noi supporremo d'ora in avanti che il segno di esponenziazione esista.

IL TIPO REALE (REAL)

Il tipo reale è quello di maggiore uso nel calcolo scientifico, ed in generale in tutti quei casi in cui "numero" non vuole dire semplicemente "numero intero". Si tratti di estrarre una radice quadrata, o di calcolare l'area di un cerchio, in tutti i casi in cui c'è da risolvere un problema che richieda appena qualcosa di più della carta e matita, il "numero reale" la fa da padrone.

Non pensare però che i numeri reali di cui si parla nei calcolatori siano i numeri reali di cui si parla in matematica: in realtà con questo nome nei linguaggi di programmazione si indicano semplicemente i "numeri decimali", non i numeri decimali "con infinite cifre decimali" (periodici o non) della matematica. Del resto nessun calcolatore potrebbe permettere la rappresentazione di un numero decimale con **infinite** cifre decimali!

La precisione con cui si può definire un numero reale dipende dal calcolatore che si usa e dal tipo di rappresentazione usato: è normale avere delle rappresentazioni dei numeri nella cosiddetta "singola precisione" (8 cifre esatte) o nella cosiddetta "doppia precisione" (16 cifre esatte). La maggioranza dei calcolatori usa questi due tipi di rappresentazione: tieni presente però che questa non è una regola tassativa.

Altri parametri da tenere presenti sono i limiti degli ordini di grandezza entro i quali il calcolatore può operare: anche qui si possono dare solamente delle regole approssimative, dipendendo questi limiti dal calcolatore che si usa. E' però abbastanza normale che un calcolatore permetta di calcolare con numeri non più grandi di 10^{38} e non più piccoli di 10^{-38}. Sembrano numeri molto grandi da un lato e straordinariamente vicini allo zero dall'altro, ma sarà bene notare che è molto facile superare numeri di questo genere nel calcolo scientifico, in cui magari capita di ottenere dei numeri "ragionevoli" come prodotto di numeri grandissimi per numeri piccolissimi. Ed anche qui la cosa è spesso inevitabile, dal momento che a ciò costringono i sistemi di unità di misura.

Queste limitazioni sono da tenere ben presenti, perchè il calcolatore dà una segnalazione di errore se si trova, mentre esegue un programma, ad aver a che fare con dei numeri che superino i limiti che gli sono stati imposti. Precisamente dà una segnalazione di OVERFLOW nel caso in cui un numero tenti di divenire troppo grande e di UNDERFLOW se un numero vuole diventare troppo piccolo.

In PASCAL le operazioni che sono permesse coi numeri reali sono

+ : X + R indica la somma di X ed R

– : AFG – K8 indica la differenza tra AFG e K8

/ : M / P indica il quoziente di M e P

* : TONIO * GIGI indica il prodotto fra TONIO e GIGI

** : A ** X indica il valore di A elevato alla potenza X.

Anche in questo caso il segno di esponenziazione che non esisteva nel PASCAL originale è stato introdotto praticamente in tutti i PASCAL in uso oggigiorno.

Ti ricordo che l'esponenziazione **non** serve solo per fare quadrati e cubi! Se si scrive 3 ** 0.5 questo equivalente a scrivere $\sqrt{3}$, e così via.

Oltre agli operatori che abbiamo elencato si può operare sui numeri reali anche con una serie di "funzioni standard", ed anche se non abbiamo ancora parlato di funzioni non sarà difficile capire di cosa si tratta: in realtà sono queste delle operazioni molto frequenti ed utili, che sarebbe proibitivo ottenere usando solo gli operatori di cui abbiamo parlato prima. Eccole

TABELLA DELLE FUNZIONI STANDARD

ABS (X)	è il valore assoluto del numero X
SQR (X)	dà il valore del quadrato di X (cioè X^2)
SIN (X)	dà il valore di sin x (x deve essere espresso in radianti, non in gradi)
COS (X)	dà il valore di cos x (sempre con x in radianti)
ARCTAN (X)	dà il valore di ARCTG (X) (il risultato è in radianti)
LN (X)	dà il valore di LN (X) (il logaritmo naturale o neperiano di x)
EXP (X)	dà il valore di e^X (dove "e" è la base dei logaritmi neperiani, cioè 2.7182818..)
SQRT (X)	dà il valore della radice quadrata di x

Anche in questo caso queste sono le "richieste minime": può benissimo succedere che il tuo PASCAL abbia una lista più lunga di questa, ma in ogni caso non può essere più corta.

Questo completa praticamente il quadro di ciò che si intende in PASCAL per **tipo reale**, e di come si opera con entità di questo tipo. Ancora una nota a margine: nell'operazione di **divisione** gli operandi possono essere **interi**, però in questo caso il risultato non è più intero, ma **reale**. In altri termini, supponiamo di voler fare 14 : 5 : se scriviamo

14 DIV 5 otteniamo 2 (intero)

se invece scriviamo

14/5 otteniamo 2.8 (reale).

IL TIPO BOOLEANO (BOOLEAN)

Il tipo booleano consiste di quantità che possono assumere solo due valori (con-

venzionalmente chiamati **vero** e **falso**). Con quantità di tipo booleano possiamo lavorare con degli **operatori** (AND, OR e NOT) la cui azione viene specificata dalle cosiddette "tabella di verità". Per chi non lo sapesse dirò che le tabelle della verità sono un modo molto brutale di descrivere un'operazione: si elencano tutte le possibili operazioni, coi rispettivi risultati. Ciò evidentemente si può fare soltanto quando il numero di casi possibili è limitato, e questo è precisamente il caso delle quantità booleane, che possono assumere — come già detto — solo **due** valori. Nelle tabella della verità che riportiamo qui di seguito il valore "falso" è scritto 0, ed il valore "vero" è scritto 1. Ti prego di notare che questi **non** sono in realtà dei numeri, ma dei "valori di verità". Ecco quindi le tabelle di verità:

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

A	NOT A
0	1
1	0

I valori vero e falso detti TRUE e FALSE. In quasi tutti i PASCAL odierni queste sono delle metacostanti.

Queste tabelle si leggono in questo modo (III riga della I tabella): "se A è vero e B è falso allora **A AND B** è falso", e così via.

Se hai già qualche esperienza di programmazione non troverai in tutto ciò nulla di nuovo. Un'innovazione essenziale in PASCAL è invece il fatto che viene introdotta una "relazione di valore" tra i valori di verità che una quantità booleana può assumere. Più precisamente, così come dici che 3 **è minore di** 7 puoi dire che **falso** è **minore di vero**. E poichè questa è un'innovazione di cui non si capisce subito la portata, vediamo nei dettagli le conseguenze che ne derivano.

Anzitutto chiariamo subito che le operazioni che si possono fare su delle quantità booleane non sono solo quelle che abbiamo elencate, ma parecchie di più: 16 per la precisione, contando anche delle operazioni banali, quali l'indentità e similari. Eccole tutte con le loro tabelle di verità.

Tabella 4.8.1.

FUNZIONI LOGICHE

	X	VALORI DELLE VARIABILI				NOMI DELLE FUNZIONI	SIMBOLI IN PASCAL
	Y	0 0	0 1	1 0	1 1		
1	0	0	0	0	0	ZERO	FALSE
2	$X \cdot Y$	0	0	0	1	AND	X AND Y
3	$X \cdot \overline{Y}$	0	0	1	0	INHIBIT	X > Y
4	X	0	0	1	1	IDENTITY X	X
5	$\overline{X} \cdot Y$	0	1	0	0	INHIBIT	X < Y
6	Y	0	1	0	1	IDENTITY Y	Y
7	$X \oplus Y$	0	1	1	0	EXCLUSIVE OR	X < > Y
8	$X + Y$	0	1	1	1	OR	X OR Y
9	$\overline{X + Y}$	1	0	0	0	NOR	NOT (X OR Y)
10	$X \odot Y$	1	0	0	1	EQUALITY	X = Y
11	$\overline{Y}$	1	0	1	0	NOT Y	NOT Y
12	$X + \overline{Y}$	1	0	1	1	IMPLICATION	X < = Y
13	$\overline{X}$	1	1	0	0	NOT X	NOT X
14	$\overline{X} + Y$	1	1	0	1	IMPLICATION	X > = Y
15	$\overline{X \cdot Y}$	1	1	1	0	NAND	NOT (X AND Y)
16	1	1	1	1	1	ONE	TRUE
	SIMBOLI DELLE FUNZIONI	VALORI DELLE FUNZIONI					

Con questo hai una panoramica completa delle operazioni notazioni e tabelle della verità possibili tra due quantità booleane.

Vediamo ora l'altro pezzo del problema, e cioè come si scrivono in PASCAL gli operatori relazionali. E' presto detto:

= . . . è identico a . . .

<> . . . è diverso da . . .

<= ... è minore o uguale a ...

<= ... è minore di ...

>= ... è maggiore di ...

>= ... è maggiore o uguale a ...

Ed a questo punto possiamo capire perchè è stata introdotta in PASCAL la convenzione "vero è maggiore di falso". Prendiamo la VII operazione della lista: EXCLUSIVE OR. Possiamo scriverla in PASCAL, qualora ne avessimo bisogno nel nostro programma? Certo: dall'esame della tabella della verità vediamo che il risultato è vero solo quando X e Y sono **diversi**, e quindi l'operazione può venire scritta X <> Y.

Così pure per l'operazione di implicazione (XII della lista): il fatto che la funzione sia falsa quando X è vera ed Y non lo è, implica il fatto che il valore di verità di X non può mai superare quello di Y, per cui la funzione può venire scritta X < = Y. Puoi vedere così che questa innovazione dei "valori di verità" permette in realtà di esprimere in PASCAL tutte le possibili operazioni logiche, senza esclusioni. Le espressioni esplicite delle operazioni booleane in PASCAL sono riportate nell'ultima colonna della tabella.

Per esperienza personale ti posso dire che non si tratta di un'innovazione da poco: è in realtà di estrema comodità, e permette di evitare espressioni lunghe e complicate di difficile lettura ed interpretazione.

IL TIPO DI CARATTERE (CHARACTER, ABBREVIATO IN CHAR)

Anche in questo caso non posso che darti delle linee generali dal momento che i caratteri non costituiscono un sistema standardizzato. Esistono però anche in questo caso delle richieste minime che devono essere soddisfatte da qualunque sistema. Più precisamente i caratteri devono consistere almeno di

I) le lettere dell'alfabeto

II) le cifre (digits) da 0 a 9

III) lo spazio (blank)

Una quantità che appartenga a questo tipo può quindi assumere questi "valori" che ho elencato, e su di essa si può anche operare con una serie di operatori e di funzioni particolari.

Fissiamoci per un momento sulle sole lettere dell'alfabeto: un'innovazione importante di PASCAL è che le lettere dell'alfabeto vengono considerate "ordinate in successione". Questo fatto permette immediatamente di agire sulle lettere con una serie di operazioni fondamentali (4 per la precisione)

ORD (C): questa funzione restituisce il numero cardinale corrispondente alla posizione che il carattere C occupa nell'insieme dei caratteri. Nel

nostro esempio (limitato alle lettere dell'alfabeto)

ORD (A) dà 1
ORD (B) dà 2
ORD (C) dà 3 e così via

CHR (J): Questa funzione fa il lavoro inverso della precedente. Cioè, se J è un numero intero la funzione restituisce il carattere alla posizione J. In altri termini

CHR (3) dà C
CHR (4) dà D
CHR (5) dà E e così via

PRED (C): abbreviazione di **PRED**ecessor: dato un carattere C dà come risultato il carattere precedente a C nella lista dei caratteri. Facciamo degli esempi:

PRED (C) dà B
PRED (M) dà L
PRED (S) dà R e così via

SUCC (C): abbreviazione di **SUCC**essive dà come risultato il carattere successivo al carattere "C". Anche qui qualche esempio per chiarire le cose:

SUCC (L) dà M
SUCC (Q) dà R
SUCC (S) dà T e così via

Queste funzioni non valgono solo per il tipo "carattere" (che si chiama CHAR in PASCAL), ma anche per tutti i tipi definiti dal programmatore, il quale, come vedremo, ha la possibilità di definire tipi di quantità a suo piacere, per così dire "tipi di fantasia". Altra importantissima e comodissima innovazione di questo linguaggio.

Col tipo CHAR si può operare anche in altri modi: più precisamente sono validi nel tipo CHAR **tutti** gli operatori relazionali che abbiamo visto elencati nel paragrafo precedente. In altri termini: noi possiamo associate alle lettere dalla A alla Z il loro numero ordinale (da 1 a 26, per la precisione). Ebbene: le stesse relazioni che passano fra i numeri ordinali passano anche in PASCAL fra i caratteri corrispondenti. Per esempio: A è al **primo** posto della lista (cioè il suo numero ordinale è 1) mentre C è al **terzo** posto della lista (cioè il suo numero ordinale è 3). Ebbene, così come 1 < 3 PASCAL comprende anche A < C.

Anche questa possibilità nuova è estesa ai "tipi di fantasia" definiti dall'utente. Ed è un'innovazione particolarmente apprezzata ogni volta che si ha a che fare con ordinamenti di dati: per esempio ordinamenti di nomi in ordine alfabetico e similari.

Volevo anche chiarire che gli stessi risultati si possono certamente raggiungere anche con altri linguaggi (tanto per dire: si può benissimo fare una routine che ordini dei dati alfabeticamente in FORTRAN o in BASIC), però sempre a spese di una complicazione del programma che PASCAL permette di evitare fin dall'inizio.

PROCEDURE E FUNZIONI (PROCEDURES, FUNCTIONS)

Non parleremo molto di questo argomento, che verrà rimandato àlla fine di questo libro, dal momento che programmare con la filosofia di PROCEDURES e FUNCTIONS è un po' la "summa" dell'arte del programmare. Una PROCEDURE è in realtà un sottoprogramma che conviene scrivere "a parte" ogni volta che ci si accorge che una serie di operazioni (una "procedura", appunto) viene richiesta parecchie volte in un programma. Può capitare — per fare un esempio — che ci si accorga che nel corso di un programma si debbano risolvere spesso delle equazioni di II° grado. E' allora molto comodo e funzionale scrivere una volta per tutte un sotto-programma che abbia come compito la risoluzione delle equazioni di II grado. Questo sotto-programma si chiamerà una "procedura" (PROCEDURE) che possiamo per il momento battezzare QUAD. Ebbene ogni volta che nel programma principale invocheremo (si dice proprio così) la QUAD verrà risolta l'equazione di II grado del momento. Va da sè che occorrerà in qualche modo dire a QUAD quali sono i coefficienti dell'equazione medesima, e che QUAD dovrà restituire in uscita i valori delle radici dell'equazione, informandoci inoltre se i due numeri che ci restituisce sono dei numeri reali (in questo caso l'equazione ammetteva due radici reali), o se si tratta di parte reale immaginaria di due radici complesse coniugate (è questo il caso dell'equazione che nelle Scuole Medie si dice impropriamente che "non ammette soluzioni", il tutto perchè i programmi ministeriali si ostinano a relegare gli utilissimi numeri complessi all'Università. Speriamo bene nel futuro).

Come vedi una PROCEDURA può essere qualcosa di piuttosto complesso, che in genere avrà bisogno di quantità in ingresso (input parameters) e restituirà delle quantità alla fine del suo lavoro (output parameters) quando il controllo verrà ripassato al programma principale, quello che ha appunto invocato la procedura.

Le Funzioni (FUNCTIONS) sono anch'esse delle procedure, ma hanno una particolarità. restituiscono all'uscita **un solo** valore. La ragione per cui si è ritenuto di separare questo caso particolare dal caso generale è ovvia: le funzioni hanno una enorme importanza pratica, e separarle dal contesto generale permette una programmazione più flessibile e di più facile lettura.

LE DICHIARAZIONI

Abbiamo visto che un programma in PASCAL è strutturato a blocchi, che ogni blocco è un qualcosa di autosufficiente che contiene in sè stesso tutto quanto gli è necessario, e che in particolare deve contenere esplicitamente la definizione di tutto ciò che nel blocco viene in qualche modo usato.

Così, se nel blocco si usa una costante chiamata UNO che all'inizio del blocco si vuole porre uguale a 1, e si vuole che con UNO venga **sempre** in tutto il blocco indicato il numero 1, occorre dichiarare che UNO è una costante (uguale ad 1, in questo caso). Se nel blocco si usa una procedura (magari la QUAD di cui si era già parlato nel precedente paragrafo) occorre dire che QUAD è una procedura, **non solo** ma anche in fase di definizione descrivere **completamente** 1a QUAD. Questo è molto importante per chi è abituato a programmare in FORTRAN. Normalmente infatti in FORTRAN si scrive un programma principale, il quale chiama delle subroutines,

le quali poi vengono scritte **dopo ed al di fuori del programma principale** (dopo la END, per intenderci): queste vengono poi compilate separatamente, e non hanno niente a che vedere in certo senso col programma principale (il MAIN) stesso. Proprio per questo in genere occorrono **due** programmi per costruire il "modulo eseguibile": il primo passaggio **compila** (cioè traduce il programma scritto in FORTRAN in linguaggio macchina) ed il secondo **cuce i pezzi** (cioè aggiusta indirizzi e rimandi dalle varie subroutines, che sono dopo il primo passo dei programmi indipendenti).

Niente di simile in PASCAL: un programma è un'unica unità organica, e le subroutines (cioè le procedure) fanno parte del MAIN e sono descritte completamente al suo interno. Un solo passaggio (compilazione e cucitura) è necessario per ottenere il programma eseguibile. Cosa si perde? Si perde l'abitudine di avere dei pacchetti di schede in cui sono descritte le varie subroutines necessarie al programma. Non è un gran male: ormai le schede non le usa più nessuno (personalmente alle volte ne ho un po' di nostalgia) ed i programmi vengono ormai costruiti su disco o nastro, per cui le procedure che servono possono essere facilmente recuperate ed inserite in programmi vari, o — meglio — organizzate in librerie, con grande vantaggio in chiarezza ed efficienza.

Abbiamo già visto come si dichiarano labels e costanti. Vediamo ora come si dichiarano tipi, variabili, procedure e funzioni. Ti ricordo anche che l'ordine delle dichiarazioni è tassativo, e ti ricordo la frase mnemonica, La Cosa Tre Volte Più Facile, qualora avessi la tendenza a confondere l'ordine delle dichiarazioni.

Anzitutto la dichiarazione dei tipi e delle variabili. Diciamo anzitutto che torneremo sull'argomento più avanti, quando parleremo dei tipi "di fantasia" e quindi avremo modo di riparlare di dichiarazione di tipi. Per ora basterà dire che ogni quantità suscettibile di variare all'interno del programma, ed indicata come un simbolo (identificatore) è una **variable** che va dichiarata all'inizio del programma. PASCAL insomma adotta in pieno la filosofia per cui si può parlare all'interno di un blocco solo di ciò che vi si è definito. Ecco il BNF delle dichiarazioni di variabili

```
⟨dichiarazione di variabili⟩ : : = VAR ⟨identificatore⟩
                                   {,⟨identificatore⟩}  : ⟨tipo⟩ ;
                                   {⟨identificatore⟩ {,⟨identificatore⟩} : ⟨tipo⟩};

⟨tipo⟩                       : : = INTEGER | REAL | BOOLEAN | CHAR
```

Vedremo al solito le applicazioni pratiche alla fine del capitolo.

Per quanto riguarda la dichiarazione delle procedure e delle funzioni sempre con riferimento all'ipotetica procedura che risolve le equazioni di II grado (QUAD), **non basta** dire all'inizio del programma che QUAD è una procedura, ma occorre anche specificare di che cosa si tratta. In termini di FORTRAN: le subroutines **complete** vanno messe assieme agli statements dichiarativi (DIMENSION etc.) e **prima della prima** istruzione eseguibile del MAIN. Se non conosci il FORTRAN ti dirò che in PASCAL i sottoprogrammi vanno completamente scritti ed inseriti nel programma principale, in modo tale che quando il programma ne avrà bisogno li trovi già definiti e scritti completamente.

E se un sottoprogramma ha bisogno a sua volta di altri sottoprogrammi? Niente paura: ciò che vale per il programma principale (il blocco in cui i simboli sono

chiamati **globali**) continua a valere anche per i sottoblocchi: i quali pertanto potranno a loro volta avere dichiarazioni di labels, costanti, tipi, variabili, procedure e funzioni senza nessuna differenza da ciò che si è visto valere per il programma principale: l'unica avvertenza è che ovviamente, come visto nel capitolo precedente, le definizioni sono valide solo all'interno del blocco in cui vengono effettuate.

RIASSUNTO DI QUESTO CAPITOLO

Ecco il BNF di ciò che abbiamo imparato

⟨label⟩	: : =	{⟨cifra⟩} $_1^4$
⟨dichiarazione di label⟩	: : =	LABEL ⟨label⟩ {,⟨label⟩}
⟨definizione di costanti⟩	: : =	CONST ⟨identificatore⟩ = ⟨costante⟩ ; {⟨identificatore⟩ = ⟨costante⟩ ;}
⟨costante⟩	: : =	⟨numero non segnato⟩ | ⟨segno⟩ ⟨numero non segnato⟩ | ⟨identificatore di costante⟩ | ⟨segno⟩ ⟨identificatore di costante⟩ | ⟨stringa⟩
⟨numero non segnato⟩	: : =	⟨intero non segnato⟩ | ⟨reale non segnato⟩
⟨intero non segnato⟩	: : =	⟨sequenza di cifre⟩
⟨sequenza di cifre⟩	: : =	⟨cifra⟩ {⟨cifra⟩}
⟨reale non segnato⟩	: : =	⟨sequenza di cifre⟩ • ⟨sequenza di cifre⟩ | ⟨ intero non segnato⟩ • ⟨sequenza di cifre⟩ E ⟨fattore di scala⟩
⟨fattore di scala⟩	: : =	{⟨cifra⟩} $_1^2$ | ⟨segno⟩ {⟨cifra⟩} $_1^2$
⟨segno⟩	: : =	+ | –
⟨identificatore di costante⟩	: : =	⟨identificatore⟩
⟨stringa⟩	: : =	' ⟨carattere⟩ | ⟨simbolo⟩ {⟨carattere⟩ | ⟨simbolo⟩}
⟨dichiarazione di variabili⟩	: : =	VAR ⟨identificatore⟩ {,⟨identificatore⟩} : ⟨tipo⟩ ; {⟨identificatore⟩ {,⟨identificatore⟩} : ⟨ tipo⟩} ;
⟨tipo⟩	: : =	INTEGER | REAL | BOOLEAN | CHAR

PROBLEMI SU QUESTO CAPITOLO

I) E' bene usare i labels il meno possibile: in realtà un buon programma in PASCAL non dovrebbe mai o quasi avere bisogno dei labels. Ciononostante vediamo di decidere quali dei seguenti labels sono sbagliati e perchè

I)	456	VI)	7
II)	84678	VII)	67 98

III) 8647
IV) 85X3
V) 98.67
VIII) 6 54
IX) 4444
X) 1000

II) Vediamo quali di queste dichiarazioni di labels sono errate e perchè

I) LABEL 56;
II) LABELS 56,67;
III) LABEL 54, 65,888 ; 87,95;
IV) LABEL 8976, 5647, 56, 11, 99, 100;
V) LABEL 1, 3, 4, 10
VI) LABEL 34; 56; 75; 67;
VII) LABEL 9, 89, 87865, 100000;
VIII) LABEL 4;
IX) LABELS 4, 5;
X) LABEL 5, 6.7, 78;

III) Quali di queste costanti sono scritte in modo errato? e perchè?

I) .35
II) PIGRECO
III) –PIGRECO
IV) 73.35
V) 0.678
VI) –7CDT
VII) '–7CDT'
VIII) 3.56 E – 67
IX) .78 E + 1
X) 9500000 E–101
XI) 4 E –34
XII) 4.0 E 55

IV) Decidi quali di queste dichiarazioni di costanti sono errate e perchè

I) CONST UNO = 1; DUE = 2; PIGRECO = 3.14.1598;
STRINGA = 'UN LIBRO SU PASCAL';
II) CONST TRE = 3; CENTO = 1.0 E 2
III) CONST AZ2 = 43; AZ3 = 43 E–2; 3AZ = 'GIGETTO';
RAGAZZA = 'MARIA LUISA';
IV) CONSTANT UNO = 1;
V) CONST DIECI = 10.0;
VI) CONST UNO = 1; MENOUNO = –UNO; SETTE = 7.0;
RAGGIO = 6378.0 {RAGGIO TERRESTRE IN KM} ;
VII) CONST C = 3.0 E8 {VELOCITA' DELLA LUCE IN M/S} ;
FDG = 7.8; N = 0.32 E23 NUMERO DI AVOGADRO ;
VIII) CONST UNO, DUE, TRE = 1, 2, 3;
IX) CONST MILLE = 1.0E–6;
X) CONST CESARINO = 3; CESARINONE = 5;
XI) CONST {NUMERO DI AVOGADRO} N = 6.02E23;
XII) CONST N = {NUMERO DI AVOGADRO} 6.02E23;
XIII) CONST N = 6.0 {NUMERO DI AVOGADRO} 2E23;

V) Decidi dove sono gli errori nelle seguenti dichiarazioni di variabili

I) VAR J, K, M :REAL; ALIVE,DEAD: BOOLEAN;
II) VAR A, B, C, D: INTEGER; J,K,L: REAL;
III) VAR A,BB,CC: REAL
I,J,K: CHAR;

IV) VAR GHR4, H678, 4FTG: REAL;
TIZIO, CAIO, SEMPRONIO : CHAR;

V) VAR TIZIO, CAIO, SEMPRONIO, SEMPRONIONE:CHAR;

VI) Queste sono delle dichiarazioni complete, selezionate da alcuni veri programmi, e nelle quali ad arte sono stati inseriti degli errori. Cercali.

I) CONST EPS = 1E–14;
VAR X,S,SX,T : REAL; I,K,N : INTEGER;

II) VAR X,Y : REAL; J.N : INTEGER;
CONST D = 0.0625; A = 32; H = 34; LIM = 32;

III) CONST N = 20
VAR I,U,V,MIN,MAX : INTEGER;

IV) VAR SIEVE,PRIMES: REAL;
CONST N = 10000;
VAR NEXT,J: INTEGER;

V) CONST LENGTH,MAXBIT,W = 59, 58, 84;
VAR J,K,Y: INTEGER
EMPTY: BOOLEAN

VI) CONST TRE:3 ; DUE:3;
VAR J,K,L = INTEGER; A,B,C = REAL;

VII) Scrivi 10 dichiarazioni di labels a piacere e poi confrontale con la sintassi prevista, e cerca eventuali errori.

VIII) Supponi di dover fare dei calcoli per un atterraggio sulla Luna. Hai bisogno delle seguenti quantità:

Raggio della Terra: 6378 km
Distanza Terra-Luna: 384403 km
Raggio della Luna: 1738 km

Scrivi una dichiarazione di queste tre quantità come costanti di tipo REAL, esprimendo le quantità stesse nel Sistema Internazionale, cioè in metri.

IX) Rispondi con le due parole: che differenza c'è nell'uso dei due operatori DIV e / ?

X) Scrivi le tabella della verità per le seguenti funzioni booleane

I) $X \cdot \overline{Y}$
II) $X \oplus Y$
III) $(X + Y) \cdot (X \cdot Y)$
IV) $\overline{X + Y}$
V) $\overline{X} \; \overline{Y}$

XI) Nella tabella delle 16 funzioni logiche scrivi accanto alle espressioni correnti le espressioni in PASCAL. Puoi scrivere queste espressioni per tutte le funzioni? Se non lo puoi fare, cos'è che ti manca?

XII) Descrivi gli operatori che si possono usare per il tipo INTEGER e quelli che si possono usare nel tipo REAL.

XIII) Descrivi gli operatori relazionali che possono funzionare fra variabili del tipo BOOLEAN. Scrivili tutti e spiegane in dettaglio il significato e la funzione.

XIV) Dì di che tipo sono e quanto valgono i valori delle funzioni seguenti:

ORD ('H') ; CHR (19) ; PRED ('F') ; ORD ('U') ; CHR (24) ; SUCC ('G')
PRED ('0') ; ORD ('T') ; ORD ('Q') ; PRED ('T')

XV) Nel problema VIII) supponi di aver bisogno delle seguenti variabili:

- distanza percorsa dal veicolo spaziale fra la Terra e la Luna, espressa in m
- tempo impiegato a percorrere questa distanza in secondi

Temperatura minima e massima registrate nel viaggio di andata e ritorno fra la Terra e la Luna (in gradi Kelvin)
Tutte queste variabili dovrebbero essere di tipo REAL. Inoltre hai anche bisogno di

- numeri di giri compiuti dal veicolo spaziale attorno alla Luna. Questa di tipo INTEGER
- sapere se il veicolo è atterrato o no. Questa è un'informazione di tipo BOOLEAN

Supponi che tutte queste quantità debbano essere delle variabili, e scrivi le dichiarazioni di variabili opportune, cercando di invertarti degli identificatori appropriati.

XVI) In una dichiarazione di variabili ci possono essere **due** distinte dichiarazioni di variabili di tipo REAL? e degli altri tipi?

CAPITOLO 5

I TIPI SPECIALI E SUBRANGE

SCOPO DEL CAPITOLO

Alla fine di questo capitolo sarai in grado di sapere

... cosa si intende per **tipo scalare**

... come si possono definire dei tipi scalari

... quali sono le operazioni che si possono fare sulle variabili di tipo scalare generalizzato

... cosa si intende per tipo **subrange**

... quali sono le operazioni che si possono effettuare sulle variabili di tipo subrange

... i vantaggi di poter definire a piacere dei nuovi tipi di variabili.

I TIPI SCALARI

Si indicano con questo nome tutti i tipi che si riferiscono a dati "singoli", cioè non presi collettivamente, o, come si dice in PASCAL "strutturati". Vedremo che i dati strutturati hanno una varietà ben maggiore di quella cui uno può essere abituato da altri linguaggi di programmazione. Per quanto invece riguarda i tipi scalari, PASCAL presenta un'altra novità rilevante: oltre ai tipi standard (INTEGER, REAL, BOOLEAN, CHAR) visti nel capitolo precedente, l'utente ha la possibilità di accedere a tipi nuovi, definiti da lui.

Vediamo di che si tratta: supponiamo che occorra classificare (mese per mese) della corrispondenza o delle fatture o dei documenti qualunque. Il "buon metodo antico" richiedeva uno schedario in cui i documenti venivano conservati, mentre a ciascuno di essi veniva imposto un numero di codice, o una sigla anche abbastanza complessa nella quale potevano essere condensate parecchie informazioni. Queste informazioni, opportunamente riassunte in un codice, venivano di solito perforate su schede le quali venivano poi fatte leggere al calcolatore: il programma del calcolatore era quindi in grado di selezionare i documenti che rispondevano a certa caratteristiche.

Ritornando al caso in esame, supponiamo che le prime tre colonne delle schede contengano l'informazione del mese cui il documenti si riferisce. Così:

GEN, FEB, MAR, APR, MAG, GIU, LUG, AGO, SET, OTT, NOV, DIC

e vediamo come può cavarsela un programmatore in FORTRAN che voglia selezionare i documenti corrispondenti ad un determinato mese. Egli dovrà scrivere delle IF booleane del tipo

IF (MESE .EQ. 'MAR')   GOTO . . . . . . etc. etc.

dove nella variabile MESE occorre mettere ciò che è contenuto nelle prime tre colonne della scheda. Espressioni di questo tipo faranno essenzialmente un lavoro del genere. "se ciò che trovi nella variabile MESE è uguale alla stringa 'MAR' allora salta all'istruzione . . .".

Se però il programmatore è più smaliziato, ed ha lunghe notti insonni sulle spalle a causa del vecchio proverbio che dice "l'appetito vien mangiando", egli avrà ormai imparato che al suo programma si richiederanno ben presto altre prestazioni: magari si scoprirà che sarebbe molto utile poter separare i documenti dei mesi estivi dagli altri. O magari — trattandosi per esempio di fatture di un albergo — separare l'alta stagione dalla bassa stagione. In questo caso il programmatore dovrà fare un lavoro un po' più complesso, che possiamo schematizzare in questo modo:

I) leggere la variabile MESE

II) confrontare ciò che si è letto con le varie stringhe possibili ('GEN', 'FEB' . . .)

III) porre in una variabile di comodo (p. es. KMESE) 1 se si tratta di gennaio, 2 se si tratta di febbraio, e così via.

A questo punto richieste complicate tipo "selezionare i mesi che vanno da giugno a settembre inclusi" diverranno delle faccende semplici, in quanto potranno venir immediatamente tradotte con l'ausilio della variabile KMESE in istruzioni del tipo "selezionare i valori di KMESE maggiori o uguali a 6 e minori o uguali a 9".

In pratica quindi non abbiamo fatto altro che introdurre un numero (il numero ordinale) associato al "mese" in modo da poter effettuare su questo numero tutte quelle operazioni che non potevamo certo effettuare sulle stringhe GEN, FEB e così via. Insomma: ricorrendo ai numeri possiamo fare un certo numero di operazioni sui "mesi".

In PASCAL ciò viene evitato fin dall'inizio: è data la possibilità al programmatore di inventare dei tipi nuovi" ("di fantasia" se vogliamo), in modo da evitare il ricorso a filosofie complicate quali quella delle variabile ausiliarie. Nell'esempio che ti ho descritto prima l'utente potrebbe definirsi un nuovo tipo (magari chiamandolo MESE) il quale sarebbe un tipo nuovo: nè intero, nè reale, nè booleano, nè carattere, per intenderci. Per fare questo occorre usare una dichiarazione di tipo (type declaration) il cui BNF suona come segue:

⟨definizione di tipo⟩ : : = TYPE ⟨identificatore di tipo⟩
= ⟨identificatore⟩ {,⟨identificatore⟩} ;

⟨identificatore di tipo⟩ : : = ⟨identificatore⟩

Nei tipi che così vengono definiti le variabili possono assumere solo valori **singoli**, individuabili uno per uno (**o** gennaio, **o** febbraio, **o** marzo . . .). Vedremo che PASCAL offre delle possibilità diverse, quali quella di descrivere dei dati "in modo

collettivo" (un po' come membri di una stessa famiglia). Dati definiti in questo modo prendono il nome di "dati strutturati" (l'unico esempio nel FORTRAN o nel BASIC è la matrice, o "array") e le strutture possibili sono parecchie, anzi in un problema capita di solito che non è facile decidere la struttura più conveniente a descrivere i dati medesimi. I dati che invece possono essere individuati singolarmente prendono il nome di dati **scalari** (scalars), i cui tipi possono essere definiti dall'utente nel modo che abbiamo visto, e su cui si può operare con una serie di operazioni che vedremo fra poco.

ESEMPI ED OPERAZIONI

Torniamo ora all'esempio di una variabile che sia di "tipo extra": il mese dell'anno, appunto. Potrai definire, visto il BNF del paragrafo precedente, un tipo MESE con la seguente dichiarazione

```
TYPE MESE = (GEN,FEB,MAR,APR,MAG,GIU,LUG,AGO,SEP,OTT,NOV,DIC);
```

Questo definisce un nuovo tipo di variabile, che non è nè intera, nè reale, nè booleana, nè carattere: è appunto il "mese". Una variabile di questo tipo può assumere dodici valori distinti, poco importa cosa realmente diventino questi valori nel calcolatore: per noi questi sono descritti dai simboli che sono elencati nella dichiarazione del tipo.

Non ti dovrebbe essere sfuggita una piccola stranezza: ho messo per "settembre" la parola SEP (che fa tanto abbreviazione di "september") e non SET. Perchè? Snobismo? Anglofilia? Niente di tutto questo: il fatto è che la parola SET è una metacostante, ed ha un suo uso particolare e riservato all'interno del linguaggio, non potendosi usare all'infuori del suo uso previsto e codificato. Sarebbe stato un errore immediatamente segnalato dal compilatore usare SET per uno scopo diverso da quello autorizzato.

A parte ciò, mi permetterei di spezzare una lancia a favore dell'uso dell'inglese all'interno dei programmi: a parte la notevole apparenza di professionalità che ne deriva ci sonc altri indubbi vantaggi, quale la concisione, l'uso di parole entrate ormai nell'uso comune ("to dislay" va bene, ma **displeare** — come mi è capitato di leggere — un po' meno!) ed il fatto non trascurabile che dovendo scambiare dei programmi con qualcuno che italiano non è, non occorre fargli dei corsi accelerati all'Accademia della Crusca. Piaccia o no, l'inglese sta diventando il nuovo esperanto.

Dopo che nella fase dichiarativa (dopo aver definito labels e costanti) avremo quindi detto di aver bisogno anche di un nuovo tipo, cioè del **mese**, saremo autorizzati a dire che delle variabili (TIZIO, ABD, GIGETTO . . .) sono del tipo MESE. Ed a questo punto con queste variabili potremo operare usando le operazioni che su di queste sono autorizzate.

Sui tipi scalari (tutti) si possono applicare le seguenti funzioni

```
SUCC(X)     PRED(X)     ORD(X)
```

e penso che sarà bene chiarire con esempi ciò che succede:

```
I)   SUCC (GEN)    dà   FEB
     SUCC (APR)    dà   MAG
     SUCC (NOV)    dà   DIC
```

II)	PRED (MAR)	dà	FEB
	PRED (OTT)	dà	SEP
	PRED (AGO)	dà	LUG
III)	ORD (MAR)	dà	3
	ORD (LUG)	dà	7
	ORD (AGO)	dà	8

ed è da notare che in genere non esiste per i tipi scalari la possibilità di usare la funzione inversa di ORD. Questa regola però non è generale: ci sono delle Case che con opportuni trucchi sono riuscite a mettere nei loro PASCAL anche questa funzione.

Un'altra nota a margine è che il tentativo di ottenere PRED (GEN) o SUCC (DIC) dà – evidentemente – luogo ad una segnalazione di errore: si tenta, in questi casi, di uscire dai valori che sono stati dichiarati leciti per il tipo in questione.

Puoi anche vedere che non è necessario che sia il programmatore a costruirsi delle variabili di comodo (come la KMESE di cui avevamo parlato): in PASCAL si può ottenere il valore dell'ordinale associato al tipo in esame semplicemente con il richiamo di una funzione standard.

Un'altra serie di operazioni che è resa possibile in PASCAL deriva dal fatto che tutti gli operatori relazionali sono **validi** se applicati a tipi scalari. Supponiamo che MESE1 sia una variabile di tipo MESE: possiamo costruire delle variabili booleane così

MESE = APR oppure MESE <> LUG

la prima essendo vera quando MESE1 sarà uguale ad APR e la seconda quando MESE1 sarà diverso da LUG.

Non così immediato risultato l'uso degli **altri** operatori relazionali: cosa vorrebbe dire, infatti

MESE > MAG?

Il mistero si svela rapidamente se si pensa i tipi scalari vengono **ordinati** nel momento della loro definizione: in altri termini il calcolatore capisce che nel tipo MESE è vero che GEN **è minore di** FEB, NOV **è maggiore di** APR e così via. Insomma tra i valori che una variabile di tipo MESE può assumere passano le stesse relazioni che passano **tra gli ordinali** corrispondenti: PASCAL permette però all'utente di non ricorrere obbligatoriamente agli ordinali, quando questo non sia strettamente necessario.

Ultima particolarià da notare (e che non è valida per tutti i PASCAL del commercio) è che nel PASCAL originale il numero ordinale associato al primo elemento della lista (GEN, nel solito esempio dei mesi) vale 0 e non 1.

Riassumendo, quindi, potremo dire che con i tipi scalari è permesso l'uso di tutti gli operatori relazionali e l'uso delle funzioni ORD, SUCC, PRED.

Per il resto, i tipi scalari possono essere i più svariati: note musicali, colori dell'arcobaleno, mesi, giorno della settimana . . . Il limite è semplicemente la fantasia di chi scrive.

I TIPI SUBRANGE

Capita molto spesso, quando si usano dei tipi scalari, di imbattersi in un problema particolare: all'interno dei valori che una variabile scalare può assumere esistono dei valori in qualche modo "privilegiati" che sarebbe molto comodo poter trattare "a parte".

Spieghiamoci meglio ricorrendo al solito esempio dei mesi. Può capitare — per esempio — di avere la necessità di trattare in modo particolare i mesi estivi per varie ragioni, abbastanza ovvie se pensi a problemi del tipo fatture o corrispondenze di qualche albergo o di qualche impresa commerciale.

A questo proposito PASCAL permette di evitare il ricorso a complesse e noiose decisioni logiche, dando la possibilità di definire un nuovo tipo, associato al tipo MESE, e che potremmo chiamare MESE ESTIVO, O MESEST per brevità. Questo tipo particolare associato ad un tipo principale (un qualcosa come un "sottotipo") prende il nome di tipo SUBRANGE e nel nostro esempio avrebbe una definizione di questo genere

TYPE MESEST = GIU . . AGO;

con il BNF generale

⟨dichiarazione di tipo subrange⟩ : : = TYPE ⟨identificatore di tipo⟩ =
⟨identificatore⟩ . . ⟨identificatore⟩ ;

Una dichiarazione come quella più sopra descritta dà automaticamente la possibilità di definire delle variabili del tipo MESEST che possono assumere i valori GIU, LUG, AGO e **che si possono contemporaneamente trattare anche come variabili di tipo MESE.**

Possiamo adesso moltiplicare gli esempi: per esempio possiamo definire un nuovo tipo (subrange ai numeri interi) definito con i numeri interi che vanno da 1 a 12:

TYPE DOZZINA = 1 . . 12;

e così via.

Non sempre però è necessario dichiarare esplicitamente la introduzione di un nuovo tipo già definito. Questo compito può essere lasciato direttamente alla definizione delle variabili. Per esempio se MESEA, MESEB, MESEC fossero tre variabili di tipo MESE, che però noi sappiamo a priori poter assumere solo i valori GIU, LUG, AGO, potremo definire queste variabili direttamente come tipo subrange, tramite una dichiarazione di questo genere:

VAR MESEA, MESEB, MESEC: GIU . . AGO;

con una dichiarazione di questo tipo si otterrà il duplice scopo di definire un tipo subrange (cui però **non** viene attribuito un nome, in questo caso) e di dichiarare delle variabili che appartengono a questo tipo.

Non si può dire a priori quando convenga usare uno dei due metodi di dichiarazione: il primo è più ordinato, ed il secondo è più spiccio. Penso che la cosa migliore è che ognuno si regoli secondo il suo istinto e la sua pratica. Anche perchè questi sono mestieri che non si possono imparare semplicemente leggendo un ma-

nuale od un libro, ma si devono apprendere soprattutto **facendo programmi** e risolvendo problemi concreti. Del resto questo modo di apprendere è comune a tutti i mestieri del mondo.

RIASSUNTO DI QUESTO CAPITOLO

In questo capitolo abbiamo visto i seguenti punti

I) si possono definire dei tipi diversi da quelli standard, a volontà del programmatore

II) su questi tipi si può operare con le funzioni PRED, SUCC, ORD e con tutti gli operatori relazionali.

III) nell'ambito di un tipo scalare — qualora se ne presenti la necessità — si può definire una sequela di valori **in sequenza** come un nuovo tipo che prende il nome di "subrange" del tipo principale.

Ecco il BNF di ciò che abbiamo imparato:

⟨definizione di tipo⟩ : : = TYPE ⟨identificatore di tipo⟩
= ⟨identificatore⟩ {, ⟨identificatore⟩} ;

⟨dichiarazione di tipo subrange⟩ : : = TYPE ⟨identificatore di tipo⟩
= ⟨identificatore⟩ . . ⟨identificatore⟩ ;

⟨identificatore di tipo⟩ : : = ⟨identificatore⟩

PROBLEMI ED ESEMPI

I) Scrivi la dichiarazione di tipo per le note musicali

II) Scrivi la dichiarazione di tipo subrange per le lettere dell'alfabeto che vanno dalla L alla S

III) Non ne abbiamo esplicitamente parlato, ma comunque, secondo te avrebbe senso parlare di tipo subrange per il tipo dei numeri reali

IV) Anche di questo non abbiamo parlato esplicitamente: ha senso secondo te parlare di subrange del tipo booleano?

V) Scrivi una dichiarazione di tipo per i giorni della settimana.

VI) Con riferimento al problema I), supponendo che il primo elemento della dichiarazione sia il DO, scrivi

SUCC(RE) ; PRED(FA) ; ORD(SOL) ; ORD(SI)

VII) Sempre con riferimento al problema I) scrivi, usando gli operatori relazionali (=, <, =<, >, >=, <>) le relazioni che intercorrono fra

MI e LA ; SOL e DO ; DO e RE ; RE e SI ; SI e DO

VIII) Supponi di scrivere una dichiarazione di tipo per i giorni della settimana di

due settimane successive, chiamando LUN1 il lunedì della prima e LUN2 il lunedì della settimana successiva e così via: scrivi la dichiarazione di tipo di questo tipo di variabile, e poi scrivi la dichiarazione di tipo di una variabile che possa andare dal martedì della prima settimana solo fino a lunedì della settimana successiva.

IV) Una costante di tipo CHAR viene messa fra due apici, per esprimere il fatto che è appunto di tipo CHAR. Così 'A' indica la A, di tipo CHAR. Se poi si deve esprimere proprio l'apice, allora questo deve venir raddoppiato.
Sapendo questo scrivi la dichiarazione di tipo per una variabile che possa assumere tutti i "valori" dalla 'A' alla 'Z'. Scrivine poi un'altra per una variabile che possa assumere i valori dalla 'G' alla 'T'.

X) Scrivi in modo legale per PASCAL la stringa

ABBASSO L'INTER

Se non sei d'accordo con il contenuto filosofico della stringa ti è permesso sostituire la parola ABBASSO con VIVA.

XI) Come faresti a scrivere la dichiarazione di tipo subrange per i caratteri alfabetici non appartenenti all'alfabeto italiano?

XII) E come faresti a scrivere la dichiarazione di tipo subrange per i soli caratteri alfabetici appartenenti all'alfabeto italiano?

CAPITOLO 6

GLI STATEMENTS DI ASSEGNAZIONE

SCOPO DEL CAPITOLO

Alla fine di questo capitolo sarai in grado di

. . . sapere cos'è un fattore

. . . sapere cos'è un termine

. . . sapere cosa sono gli operatori additivi e moltiplicativi

. . . sapere cosa sono gli operatori di relazione

. . . sapere cosa sono delle espressioni semplici e delle espressioni in generale

. . . sapere cosa vuol dire uno statement di assegnazione

. . . scrivere delle espressioni complete in PASCAL

. . . scrivere degli statements di assegnazione in PASCAL

. . . scrivere uno statement composto in PASCAL

GLI STATEMENTS DI ASSEGNAZIONE

Questi sono in tutti i linguaggi, e quindi anche in PASCAL, gli attrezzi fondamentali per il calcolo. Salvo alcuni dettagli gli statements di assegnazione in PASCAL non differiscono molto da quelli degli altri linguaggi. Vediamone le caratteristiche.

Lo statement di assegnazione singolo ha la seguente sintassi

⟨statement di assegnazione⟩ : : = ⟨variabile⟩ <u>: =</u> ⟨espressione⟩

che si legge

"il valore della variabile **è sostituito dal** valore che risulta dal calcolo dell'espressione". Il simbolo di assegnazione (: =) è una metacostante e non va usato per altri scopi. E' bene ricordare esplicitamente che il simbolo : = viene usato solo in questo caso e non è da confondere in PASCAL con il simbolo =, che è un simbolo **relazionale**, usato negli statements dichiarativi, come hai visto nei precedenti capitoli, o per costruire delle variabili booleane.

Nello statement di assegnazione va ancora completato ciò che intendiamo per "espressione". Il BNF è piuttosto complesso:

⟨espressione⟩ : : = ⟨espressione semplice⟩
|⟨espressione semplice⟩ ⟨operatore di relazione⟩ ⟨espressione semplice⟩

⟨espressione semplice⟩ : : = ⟨termine⟩
|⟨espressione semplice⟩ ⟨operatore di addizione⟩ ⟨termine⟩
|⟨operatore di addizione⟩ ⟨termine⟩

⟨termine⟩ : : = ⟨fattore⟩
|⟨termine⟩ ⟨operatore di moltiplicazione⟩ ⟨fattore⟩

⟨fattore⟩ : : = ⟨variabile⟩ |⟨costante non segnata⟩
|⟨designatore di funzione⟩ |(⟨espressione⟩)
|NOT ⟨fattore⟩

⟨costante non segnata⟩ : : = ⟨numero non segnato⟩
|⟨stringa⟩ |⟨identificatore di costante⟩

⟨operatori di moltiplicazione⟩ : : = * |/ |DIV |MOD |AND

⟨operatori di addizione⟩ : : = + |− |OR

⟨operatori di relazione⟩ : : = IN |= |< |<= |<> |>= |>

⟨statements di assegnazione⟩ : : = ⟨identificatore di variabile⟩ : = ⟨espressione⟩
|⟨identificatore di funzione⟩ : = ⟨espressione⟩

Esaminiamo ora cosa tutto ciò vuol dire. Dalla prima relazione del BNF troviamo che un'espressione può essere costituita o da un'espressione semplice (che chiameremo ES1, ES2, etc.) o da **due** espressioni semplici, legate però fra loro da operatori relazionali. Posto che sappiamo come fare a scrivere un'espressione semplice (non lo sappiamo ancora!) ecco delle possibili espressioni

ES1 = ES2 oppure ES1 >= ES2

Dobbiamo ora vedere come si fa a costruire un'espressione semplice. Questa può essere costruita in vari modi:

I) o come qualcosa chiamato "termine"

II) o come un termine preceduto da un operatore di addizione (e cosa sia un operatore di addizione è definito più avanti nel BNF)

III) o come una serie di termini legati fra loro da operatori di addizione. Osservate che "operatore di addizione" viene considerato anche l'OR logico.

Supponiamo quindi di aver in qualche modo costruito dei **termini** e supponiamo di chiamarli T1, T2, T3, . . . In base al BNF ed a quanto detto più sopra, ecco alcune possibilità di costruzione di espressioni semplici

T1	(un termine solo)
T1 + T2	(due termini separati dall'operatore di addizione +)
−T5	
T4 OR T7	(anche l'operatore OR è un operatore di addizione!)

E così via: penso che la vostra fantasia possa fare il resto.

A rigore, dato il BNF di prima, dovrebbero essere possibili anche espressioni del tipo

T3 + T4 OR T5

ma vedremo che ci sono delle restrizioni circa il fatto che in un'espressione i **tipi** dei termini non possono essere arbitrari. Nell'ipotetica espressione di più sopra T4 e T5 dovrebbero essere evidentemente dei termini di tipo BOOLEAN, ed a questo punto è evidente che T3 **non** può essere BOOLEAN (perchè non ha senso **sommare** quantità di tipo logico), nè di altro tipo, (in quanto non è possibile sommare a quantità di tipo BOOLEAN delle quantità di altri tipi). E' per questa ragione che espressioni come la precedente non hanno diritto di cittadinanza.

Continuiamo ad aprire scatole cinesi, e vediamo come si può definire un **termine.** Questo è definito

I) o come un **fattore**

II) o come una serie di fattori legati fra loro da **operazioni di moltiplicazione.** Sarà bene notare esplicitamente che gli operatori di moltiplicazione **non** sono la semplice moltiplicazione, ma anche DIV, MOD, AND

Al solito supponiamo che F1, F2, F3 . . . siano dei fattori. Ecco una serie di **termini** possibili

F1 DIV F2
F3 MOD F1 DIV F6
F1 / F4 * F6
F7 AND F5 AND F2 e così via.

Sono evidenti le analogie con le regole algebriche (notate bene: **analogie!**)

fattore ⟶ lettera o numero
termine ⟶ monomio
espressione semplice ⟶ polinomio

Il gioco delle scatole cinesi è ormai finito: il BNF ci informa alla fine che un fattore può essere una costante, o una variabile, o anche un'intera espressione **purchè chiusa in parentesi** o una funzione o un fattore preceduto da **NOT.**

Possiamo a questo punto tirare le somme e vedere dove siamo arrivati. Anzitutto abbiamo già stabilito una gerarchia delle operazioni: si calcolano **prima** i fattori, **poi** i termini, **poi** le espressioni semplici, **poi** le espressioni. Ciò vuol dire che nel caso in cui siamo in presenza di espressioni complesse si inizia ad effettuare i calcoli dalle parentesi **più interne** (vi ricordo che un fattore può anche essere costituito da una espressione racchiusa in parentesi). Fra tutti gli operatori la priorità più alta spetta al NOT logico. Dopo di questo vengono eseguite tutte le operazioni *,/DIV, MOD, AND. Se il PASCAL che hai a disposizione ha anche l'operatore di esponenziazione (**) questa operazione viene eseguita **prima** delle operazioni di moltiplicazione e subito **dopo** il NOT logico. Dopo le operazioni di moltiplicazione vengono eseguire le operazioni di addizione : + , – , OR.

Data l'importanza di ciò che veniamo dicendo, non sarà male rifare il cammino inverso, aiutandoci con esempi:

un fattore può essere definito nei seguenti modi

I) una variabile

X ABC BEPPE W78 SKYLAB

II) una costante **non** segnata

3 5.6 3.89-23 'ATALANTA'

III) un designatore di funzione

SQRT(AREA) SIN(LATID/57.2958)

IV) un fattore preceduto da NOT

NOT MALATO NOT PROMOSSO

V) un'espressione in parentesi

(SQRT (X) +3.47)

Con i fattori possiamo costruire i termini, a patto di legarli con operatori moltiplicativi. Ecco ad esempio alcuni termini:

```
5.78 * SQRT (XX) / (3.67 + JUVE);
(A + B) DIV (4-HGF)
STUDENT AND NOT SICK AND MARRIED
```

Alla fine possiamo ottenere delle espressioni semplici, legando i vari termini con delle operazioni in addizione, ed infine, legando assieme espressioni semplici otteniamo le espressioni.

ALCUNE OSSERVAZIONI ED ALCUNI CONSIGLI

I) Quando ci si trova di fronte a parecchi operatori moltiplicativi questi vengono presi in considerazione nell'ordine in cui appaiono. Ciò non sempre porta a delle espressioni facilmente leggibili anche se perfettamente comprensibili al calcolatore. Supponiamo per esempio di voler scrivere in PASCAL la formula

$$\frac{a}{b \cdot c}$$

Potremmo scrivere benissimo A / B / C: il risultato sarebbe che il calcolatore prende A e lo divide per B, quindi prende il risultato dell'operazione e lo divide ancora per C. Evidentemente questo era proprio ciò che volevamo ottenere. Però penso che sarai d'accordo nel riconoscere che scrivere A / (B * C) è molto più chiaro.

II) A questo proposito non sarà male spendere due parole per incoraggiare l'uso abbondante delle parentesi, che facilitano spesso la comprensione del programma. In generale è buona norma pensare che il programma non lo facciamo per noi, ma per essere usato, capito, e magari modificato da altri, dove per

"altri" si può intendere noi stessi a distanza di qualche mese, quando ci saremo dimenticati i dettagli del problema ed i trucchi cui siamo ricorsi per risolverlo. Queste due scritture

A/J–H + L * P > K * X + G * H–RT
(A/J–H + L * P) > (K * X + G * H–RT)

portano allo stesso risultato, ma la seconda è molto più chiara della prima.

III) In uno statement di assegnazione i tipi della variabile e dell'espressione **debbono essere uguali**. Questa potrà sembrare una restrizione ed un passo indietro rispetto ai moderni compilatori FORTRAN che sono molto tolleranti in materia, mentre una volta erano rigidissimi. In realtà una volta questa restrizione era mantenuta soprattutto per poter avere dei compilatori semplici, mentre ora la si ripristina soprattutto per avere una programmazione ordinata.
E' tollerata un'unica eccezione: se la variabile è **reale** l'espressione può dare come risultato sia un REAL che un INTEGER. Attenzione che **non è lecito il viceversa.**

IV) Poichè per quanto visto PASCAL **non** accetta faccende come

N := 4.5

dove N sia INTEGER (statement che sarebbe lecito in FORTRAN e che assegnerebbe ad N il valore 4, INTEGER), viene spontanea la domanda di come possa convertire in INTEGER il risultato di un'espressione REAL. Fra l'altro è un problema che capita spesso.
Vengono in aiuto le "funzioni speciali" di PASCAL che sono le uniche autorizzate a collegare fra loro variabili di tipo diverso. Queste sono

ROUND(X)	dato X di tipo REAL (supponiamo 4.56) restituisce un valore INTEGER arrotondando X all'intero più vicino (si otterrebbe 5 nel caso in esame)
TRUNC(X)	dato X di tipo REAL (e supponiamo che sia ancora 4.56) **tronca** la parte decimale e restituisce l'intero risultante (cioè 4 nel caso in esame)
ODD (X)	dato un X di tipo INTEGER restituisce un valore BOOLEAN, che è **vero** se X è dispari, altrimenti è **falso.**

Vedi che con l'aiuto delle funzioni speciali ci sono tutte le possibilità di effettuare una varietà di operazioni, senza per questo rinunciare a chiarezza ed ordine, che sono fra i pregi maggiori di questo linguaggio.

LO STATEMENT COMPOSTO

Concludiamo questo capitolo con una novità per coloro che conoscono il FORTRAN, ma non per chi conosce ALGOL o PL/1 e simili): lo statement composto.

Il BNF di questo tipo di statement è presto scritto

⟨statement composto⟩ : : = BEGIN ⟨statement⟩ {; ⟨statement⟩} END

e non avrebbe bisogno di lunghi commenti se non fosse per il fatto che tutto ciò che sta entro le parole BEGIN . . . END viene trattato nel programma come se fosse **un unico statement.**

Le parole BEGIN . . . END assumono insomma un po' il ruolo di "parentesi logiche" per così dire. Un po' come le parentesi vere nel caso delle espressioni che abbiamo visto nel paragrafo precedente.

Vedremo nei prossimi capitoli che ciò diviene molto importante nel flusso logico del programma, dal momento che fra le parole BEGIN ed END può trovare posto veramente un intero programma ed anche perchè gli statements composti possono essere soggetti completamente al processo di "nesting", cioè possono essere messi uno dentro l'altro, senza alcuna limitazione. Con ciò PASCAL offre al programmatore la possibilità di avere in realtà un insieme di statements anche estremamente complesso, che però nel flusso logico del programma principale verrà trattato come uno statement singolo.

Questa non è però una novità di PASCAL: questo tipo di possibilità è stata introdotto per la prima volta nell'ALGOL, già dal 1958. E ti assicuro che questa è una enorme facilitazione, il cui uso diverrà sempre più evidente man mano che andremo avanti col linguaggio nei prossimi capitoli.

RIASSUNTO DI QUESTO CAPITOLO

⟨statement di assegnazione⟩ : : = ⟨variabile⟩ : = ⟨espressione⟩

⟨espressione⟩ : : = ⟨espressione⟩
| ⟨espressione semplice⟩ ⟨operatore di relazione⟩ ⟨espressione semplice⟩

⟨espressione semplice⟩ : : = ⟨termine⟩
| ⟨espressione semplice⟩ ⟨operatore di addizione⟩ ⟨termine⟩
| ⟨operatore di addizione⟩ ⟨termine⟩

⟨termine⟩ : : = ⟨fattore⟩
| ⟨termine⟩ ⟨operatore di moltiplicazione⟩ ⟨fattore⟩

⟨fattore⟩ : : = ⟨variabile⟩ ⟨costante non segnata⟩
| ⟨designatore di funzione⟩ | (⟨espressione⟩)
| NOT ⟨fattore⟩

⟨costante non segnata⟩ : : = ⟨numero non segnato⟩
| ⟨stringa⟩ | ⟨identificatore di costante⟩

⟨operatori di moltiplicazione⟩ : : = * | / | DIV | MOD | AND

⟨operatori di addizione⟩ : : = + | – | OR

⟨operatori di relazione⟩ : : = IN | > | > = | = | < = | < | <>

⟨statement di assegnazione⟩ : : =
⟨identificatore di variabile⟩ : = ⟨espressione⟩
| ⟨identificatore di funzione⟩ : = ⟨espressione⟩

⟨statement composto⟩ : : = BEGIN ⟨statement⟩ {; ⟨statement⟩} END ;

PROBLEMI ED ESEMPI

I) Controlla se le seguenti espressioni sono scritte correttamente

```
D :=SQR(B)-4.0 * A * C;
A :=B*SIN (ALFA) / SIN (BETA);
A :=(B*SIN (ALFA) / SIN (BETA)
```

II) Detto P il perimetro di un triangolo ed A, B, C i suoi lati, la cosiddetta "formula di Erone" stabilisce che l'area del triangolo è data da

$$S = \sqrt{\left(\frac{P}{2}\right)\left(\frac{P-A}{2}\right)\left(\frac{P-B}{2}\right)\left(\frac{P-C}{2}\right)}$$

Scrivi un'espressione per calcolare l'area di un triangolo con la "formula di Erone".

III) Scrivi l'espressione per calcolare il volume della sfera dato che ne sia il raggio.

IV) Fai un elenco degli operatori additivi

V) Fai un elenco degli operatori moltiplicativi

VI) Fai un elenco degli operatori relazionali

VII) Riprendi ora l'elenco completo di tutte le funzioni booleane del capitolo IV. Scrivi di tutte l'espressione in PASCAL. Puoi farlo, ora? E — se sì — cosa ti permette di farlo adesso, e cosa di impediva di farlo nel capitolo IV?

VIII) Dati due lati e l'angolo compreso di un triangolo la "formula di Carnot" permette di calcolare il lato restante. La formula è

$$a^2 = b^2 + c^2 - 2\,b\,c\cos(\alpha)$$

Scrivi un'espressione in PASCAL che calcoli A (e non il suo quadrato, come nella formula di Carnot), a partire da B e C e dall'angolo ALFA, opposto al lato A, e quindi compreso tra B e C. Supponi una prima volta che ALFA sia espresso in radianti, poi che sia espresso in gradi e sottomultipli decimali (decimi e centesimi di grado).

IX) Con riferimento alla domanda precedente, scrivi una dichiarazione delle costanti e delle variabili di cui hai avuto bisogno per la tua espressione.

X) Scrivi delle dichiarazioni di tipo per i giorni della settimana e per i mesi dello anno. Supponi poi che non si lavori il sabato e la domenica, oltre a tutto il mese di agosto. Definisci una variabile booleana che si dica se un determinato giorno è lavorativo o no. E scrivi alla fine un'espressione BOOLEANA che dica se un certo giorno della settimana di un certo mese è lavorativo o no. Hai bisogno di dichiarare altre variabili oltre a quelle qui suggerite? E di quali tipi?

XI) Nella scala cromatica estisono le seguenti note

DO,DODIESIS,RE,REDIESIS,MI,FA,FADIESIS,SOL,SOLDIESIS,LA,LADIESI,SI

L'intervallo fra due note consecutive viene anche chiamato un semitono, e si dice che fra due note c'è un accordo di terza o di quinta se fra loro c'è un intervallo rispettivamente di a 4 o di 7 semitoni.
Scrivi tutte le dichiarazioni di tipi, costanti e variabili di cui hai bisogno, e due funzioni booleane che dicano se fra due note c'è o no un intervallo di terza o di quinta.

XII) Supponiamo che tu debba calcolare sia la superficie che il volume di una sfera. Scrivi le dichiarazioni di tutte le quantità di cui avrai bisogno. Infine scrivi le espressioni per il calcolo di superficie e volume, in modo che queste risultino formare uno statement composto.

XIII) Scrivi uno statement composto per il calcolo delle **due** radici reali di un'equazione di II grado (supponendo che il discriminante sia positivo). Scrivi anche tutte le definizioni e dichiarazioni delle quantità di cui hai bisogno.

CAPITOLO 7

GLI STATEMENTS DI RIPETIZIONE

SCOPO DEL CAPITOLO

Alla fine di questo capitolo sarai in grado di

... sapere cos'è uno statement ripetitivo ed a cosa serve

... sapere cos'è uno statement FOR

... usare lo statement FOR in PASCAL

... sapere cosa sono gli statements WHILE e REPEAT

... usare gli statement WHILE e REPEAT

... scrivere dei pezzi di programma che usano espressioni e statements iterativi

GLI STATEMENTS RIPETITIVI

La possibilità di ripetere una serie di operazioni per un certo numero di volte deciso a priori o anche durante l'elaborazione del programma stesso, è uno dei punti di forza dei linguaggi di programmazione.

Nei linguaggi "a basso livello" (come per esempio in tutti gli "assemblers", o linguaggi macchina se vuoi), è responsabilità del programmatore costruire lo statement iterativo, detto anche "loop" (anello, circolo). In genere ciò viene fatto con una serie di istruzioni che posso così sintetizzare:

I) viene introdotta una variabile di servizio (detta anche contatore) che d'ora in avanti chiameremo J, un valore iniziale di questa variabile (JIN), un valore finale (JFIN) ed un incremento (DJ).

II) all'inizio di ogni ciclo J viene aumentata (può anche essere diminuita, nel caso che DJ sia negativo!) della quantità DJ.

III) il ciclo quindi viene eseguito, e tutte le operazioni che in esso sono specificate vengono effettuate.

IV) alla fine del ciclo il valore di J quale risulta dopo l'incremento di cui al punto II) viene confrontato con JFIN.

V) Se JFIN non è stato superato il programma ritorna al punto II)

VI) Se JFIN è stato superato il programma esce dal ciclo e prosegue con le altre istruzioni.

Capirai che una tale serie di operazioni non è certo semplice da descrivere al calcolatore, anche se in fondo bisogna riconoscere che dopo un po' tutta questa filosofia diviene molto automatica e spontanea a chi lavora, per cui tutto ciò viene fatto quasi "senza pensarci". In ogni caso è sempre una bella noia che allunga notevolmente lavoro e programma. Ovvia quindi la necessità di scorciatoie.

In FORTRAN la scorciatoia è lo statement "DO", la cui sintassi è molto semplice

DO 73 J = JIN, JFIN, DJ

vuol dire in sostanza

"fai (DO, appunto) tutto ciò che c'è fra qui e lo statement 73 incluso, la prima volta dando a J il valore JIN, poi JIN+DJ, poi . . . finchè non viene raggiunto il valore JFIN. A questo punto esci dal ciclo eseguendo lo statement successivo".

PASCAL offre a chi programma una maggiore flessibilità senza rinunciare al pregio maggiore del FORTRAN, cioè alla concisione. In PASCAL gli statements iterativi sono 3 da usare a seconda delle circostanze. Prendono il nome di FOR, WHILE e REPEAT, e li vedremo uno per uno nei prossimi paragrafi.

LO STATEMENT FOR

Questo statement si usa di preferenza nei casi in cui sia noto **a priori** il numero di iterazioni che si devono eseguire.

Possiamo fare alcuni esempi: moltiplicare fra di loro due matrici di dimensioni note; oppure controllare e stampare le prime 70 fatture di un archivio; oppure estrarre a caso 150 cartelle cliniche di pazienti cardiopatici dall'archivio di un ospedale, per fare un'indagine statistica.

In tutti questi casi lo statement FOR è il più indicato. Il BNF assume due possibili forme

⟨statement FOR⟩ : : = FOR ⟨variabile di controllo⟩
: = ⟨valore iniziale⟩ TO ⟨valore finale⟩
DO ⟨statement⟩ ;

E' possibile anche l'altra forma:

⟨statement FOR⟩ : : = FOR ⟨variabile di controllo⟩ : = ⟨valore iniziale⟩
DOWNTO ⟨valore finale⟩ DO ⟨statement⟩

⟨variabile di controllo⟩ : : = ⟨identificatore⟩

⟨valore iniziale⟩ : : = ⟨espressione⟩

⟨valore finale⟩ : : = ⟨espressione⟩

L'uso dello statement FOR può venir spiegato molto semplicemente in questo modo: lo statement dopo la **DO** (e ti ricordo che in PASCAL per **statement** si può anche intendere un intero programma, dal momento uno "statement" può essere anche uno "statement composto", cioè un insieme anche molto lungo e complesso di istruzioni purchè comprese fra le parole BEGIN . . . END) viene eseguito dando alla variabile di controllo dei valori che partono dal valore iniziale ed arrivano fino al valore finale. Ad ogni ciclo questi valori sono incrementati (è la prima versione, il ciclo con la parola TO) di 1, oppure decrementati (ed è la seconda versione, con la parola DOWNTO) di 1. Non sono permessi incrementi o decrementi diversi da questi.

Questo a prima vista può sembrare una limitazione pesante: in pratica obbliga ad usare sempre ed in ogni caso incrementi (o decrementi) **di una sola unità.** E chi ha un po' di esperienza di programmazione sa quanto sia comodo e spiccio usare l'indice di un'iterazione che viene incrementato ogni volta di — diciamo — 5 per esaminare dei dati di una lista, prendendone, appunto, uno ogni 5. Ebbene: anche se è comodo e spiccio è una pessima pratica di programmazione. Infatti la variabile di controllo del ciclo ha una funzione tutta speciale (quella di controllare il ciclo, appunto) ed è bene non usarla per altri scopi. Già molti compilatori vietano esplicitamente la **modifica** della variabile di controllo: PASCAL va più in là con la restrizioni ed incoraggia esplicitamente la pratica di lasciare la variabile di controllo semplicemente alla sua funzione.

Vediamo ora qualche esempio di FOR. Evidentemente gli esempi sono inventati e non hanno particolari significati: servono solamente a far vedere la sintassi dello statement in modo più chiaro.

```
FOR N := 1 TO 6 DO X := X + 1;
FOR JJJ := 56 TO 84 DO   TIZIO := TIZIO * 2;
```

Ed ecco anche un esempio di FOR in cui lo statement controllato dalla DO è uno statement composto:

```
FOR COLOR := 1 TO 6 DO
BEGIN
   XX := YY-1; CAIO := GRACCO-5;
   MUZIO := 5.789 * SCEVOLA;
END;
```

Oppure, usando un FOR in cui l'indice **cala** ad ogni iterazione:

```
FOR NOTA := 12 DOWNTO 1 DO
BEGIN X := S-5; W := SAX +7.65 E-5; END;
```

E dal BNF dovrebbe esserti evidente che i limiti fra i quali può muoversi la variabile di controllo non è necessario siano delle costanti. Tutto ciò che gli si richiede è che siano degli "identificatori". Quindi per esempio questa FOR va bene, anche se non piacerà a tutti.

```
FOR LAZIO := SERIEA   DOWNTO  SERIEB  DO ......
```

La quale FOR ci pone un problema. Non sportivo, s'intende. Finora abbiamo parlato sempre di variabili di controllo che assumevano valori interi, i quali potevano

scalare di 1. Quando però ci troviamo di fronte ad uno statement come quello precedente, è lecito chiedersi se LAZIO, SERIEA e SERIEB siano variabili di tipo intero obbligatoriamente, o se non siano possibili altri tipi sia per la variabile di controllo che per i suoi limiti.

In effetti è così. Sono permessi **tutti** i tipi scalari (anche quelli "di fantasia", dunque) con la sola eccezione dei tipi booleano e reale. Se ci pensi un attimo questa restrizione è piuttosto ovvia, fra l'altro. Permettere cicli con una variabile booleana significherebbe fare "molto rumor per nulla": la variabile booleana può infatti assumere solo **due** valori. Il perchè del fatto che sia proibito usare delle variabili di controllo **reali** è un po' più complesso da capire, ma non troppo, ed anzi ci porge l'occasione per impare qualcosa di nuovo.

Il fatto è che nel calcolatore le variabili di tipo intero hanno una configurazione di bits ben precisa ed individuabile: ed ogni operazione di confronto permette facilmente di discriminare fra le varie possibilità. Non è possibile che il calcolatore sbagli quando deve decidere se un certo numero è uguale a 2 (intero) oppure no. Diverso è il caso dei numeri reali, che in genre sono risultati di calcoli precedenti e che vengono dati dal calcolatore con 8 (tipicamente) cifre significative. Così a tutti gli effetti **dei nostri calcoli** un numero come 1.9999999 vale praticamente 2. Non però per il calcolatore che dovesse decidere se questo numero è **uguale** a 2 (reale) oppure no. Infatti sicuramente lo troverebbe diverso.

E' per questa ragione che il tipo REAL è escluso da dei confronti che si basano sull'uguaglianza. Ed è bene che tu ricordi che questo è un tipo di limite che esiste comunque per questo tipo di variabili. In altri termini è sempre molto rischioso fidarsi dell'uguaglianza fra due numeri reali. L'uguaglianza è una richiesta molto stringente ed è da evitare di usare il confronto fra due numeri reali, che possono fluttuare lievemente per ragioni di arrotondamento, come unico mezzo decisionale per far prendere al programma una strada piuttosto che un'altra.

Supponiamo ora che CH sia una variabile di tipo CHAR. Così pure ELLE e ZETA che supporremo proprio essere i nomi dei caratteri 'L' e 'Z'. In altri termini nel nostro programma immaginario dovrebbero esserci statements del tipo

```
VAR CH,ELLE,ZETA:CHAR;
......
ELLE :='L';    ZETA :='Z';
......
```

A questo punto sarebbe possibile una FOR di questo tipo

```
FOR CH :=ZETA DOWNTO ELLE DO
BEGIN
......
END;
```

e tutto ciò che sta nello statement composto verrebbe eseguito una prima volta dando a CH il valore 'Z', poi il valore **precedente** a Z (cioè 'Y'), poi il valore ancora precedente (cioè 'X') e via via fino a raggiungere la lettera 'L'.

Gli esempi si possono moltiplicare coi tipi definiti dallo stesso programmatore:

così possiamo avere delle FOR che operano su variabili di controllo tipo note musicali, o colori o altro.

Un'ultima nota: supponiamo di avere uno statement di questo tipo:

FOR K := 4 TO 3 DO. . . . . .

in cui — in altri termini — **il limite superiore sia più piccolo del limite inferiore.** Condizioni di questo tipo si possono presentare magari per errori di programma durante l'esecuzione del programma stesso. In questo caso lo statement che viene dopo la DO **non** viene eseguito.

Questo va notato esplicitamente a beneficio di chi di voi fosse abituato al FORTRAN: infatti in questo linguaggio gli statements pertinenti all'iterazione vengono eseguiti sempre almeno una volta. PASCAL invece (molto più razionalmente) effettua il controllo su K **prima** di iniziare ad eseguire lo statement (semplice o composto) della DO, e pertanto, nell'esempio di prima, se si trova che K è maggiore del limite superiore lo statement iterativo finisce prima ancora di essere eseguito.

GLI STATEMENTS WHILE E REPEAT

Questi statements hanno funzioni analoghe e vengono usati quando non è noto a priori il numero delle iterazioni da eseguire, come quando invece si usano le FOR.

La condizione che decide se il programma deve continuare a eseguire le istruzioni del ciclo o se deve smettere e procedere con il resto delle istruzioni, è invece una variabile logica che — fino a che è vera — costringe il programma a continuare nel ciclo. Appena diviene falsa il ciclo finisce e si continua col resto del programma.

Il BNF di queste due istruzioni è semplicissimo: detto per inciso le istruzioni sono molto simili l'una all'altra e spesso è solamente questione di gusti e di preferenze l'uso dell'una o dell'altra istruzione. Ecco quindi il BNF

⟨statement WHILE⟩ : : = WHILE ⟨espressione booleana⟩
DO ⟨statement⟩ ;

⟨statement REPEAT⟩ : : = REPEAT ⟨statement⟩ {; ⟨statement⟩}
UNTIL ⟨espressione booleana⟩ ;

In inglese questi statements sono estremamente chiari: tenete presente che WHILE e UNTIL sono quasi sinonimi. "While" vuol dire "mentre", usato qui nel senso di "fin tanto che". "Until" vuol dire "fino a che". Col che il primo statement suona all'incirca così

"fin tanto che l'espressione booleana è vera esegui (DO) lo statement in oggetto (statement che può essere anche composto, s'intende)".

Ed il secondo suona così:

"ripeti (REPEAT) tutti gli statement **fino a che** l'espressione booleana di cui in oggetto **non diventi** vera".

Detto e spiegato così non v'ha chi non veda che i due statements suonano maledettamente simili. Tanto che si può pensare che siano in pratica la stessa cosa, detta in due maniere diverse. In certo senso è così. Le differenze infatti sono minime, e dovute forse più a criteri di leggibilità che ad altro. Comunque **ci sono** delle differenze ed è bene chiarirle una per una

I) mentre nello statement WHILE si può eseguire **un solo** statement nel ciclo, nell'altro se ne possono eseguire quanti se ne vuole. Non è che ciò voglia dire molto, dato che si può sempre usare uno statement composto. In ogni caso ci si risparmia una BEGIN . . . END usando lo statement REPEAT.

II) Nello statement WHILE l'espressione booleana viene calcolata **prima** di eseguire lo statement del ciclo. Ne consegue che se questa è falsa il ciclo non viene eseguito.

III) Nello statement REPEAT l'espressione BOOLEANA viene calcolata **dopo** aver eseguito le istruzioni del ciclo: quindi nel caso questa fosse vera fin dall'inizio le istruzioni del ciclo vengono eseguite almeno una volta.

Con il che penso che la parte teorico-descrittiva sia finita, salvo la nota (che è da considerarsi piuttosto ovvia) che gli statements eseguiti in un ciclo possono **a loro volta** essere degli statements iterativi: in questo caso le iterazioni vengono racchiuse una dentro l'altra ("nesting") allo stesso modo delle scatole cinesi o delle bambolotte russe.

E passiamo a qualche esempio esplicito di WHILE e REPEAT.

ALCUNI ESEMPI

Anche questi esempi sono completamente "di fantasia", e ci servono più che altro come un riassunto di ciò che abbiamo visto finora.

```
WHILE A>6 DO
  BEGIN
    X :=T +5;          HJK := TIZIO–3;    GIGI :=BEPPE *4
  END;

WHILE  ENTER<0     AND LAZIO=–1  DO  SERIEA :=0;

REPEAT
  A :=B +5;   C :=5–(G*J–7.89);   B :=LMO –0.78
UNTIL  S<7  AND 5>K  OR  GHT  <>  LMO;
```

RIASSUNTO DI QUESTO CAPITOLO

⟨statement FOR⟩ : : = FOR ⟨variabile di controllo⟩
:= ⟨valore iniziale⟩ TO ⟨valore finale⟩ DO ⟨statement⟩ ;

⟨statement FOR⟩ : : = ⟨variabile di controllo⟩
: = ⟨valore iniziale⟩ DOWNTO ⟨valore finale⟩ DO ⟨statement⟩ ;

⟨variabile di controllo⟩ : : = ⟨identificatore⟩

⟨valore iniziale⟩ : : = ⟨espressione⟩

⟨valore finale⟩ : : = ⟨espressione⟩

⟨statement WHILE⟩ : : = WHILE ⟨espressione booleana⟩
DO ⟨statement⟩ ,

⟨statement REPEAT⟩ : : = REPEAT ⟨statement⟩ {; ⟨statement⟩}
UNTIL ⟨espressione booleana⟩ ;

PROBLEMI ED ESEMPI

I) Vuoi ottenere la somma dei primi N numeri interi (non sai come venga fornito N al programma: supponi che qualcuno in qualche modo glielo fornisca). Scrivi uno statement iterativo per ottenere questo risultato, e completalo con le parti dichiarative opportune.

II) In matematica si chiama "fattoriale di N" (e si indica con N!) il numero

$$F = 1 \cdot 2 \cdot 3 \ldots . (N-1) \cdot N$$

Scrivi uno statement iterativo per il calcolo di F. Al solito completa il tutto con dichiarazioni e definizioni necessarie.

III) Nel calcolo di un fattoriale può facilmente capitare di ottenere numeri straordinariamente grandi, che il calcolatore non accetta. Modifica il programma precedente in modo da non permettere che il calcolo venga effettuato quando si arriva a numeri più grandi di 10^{35}. Quale tipo di statement iterativo ti conviene usare in questo caso?

IV) Spiega la differenza che passa fra gli statements WHILE e REPEAT.

V) Scrivi due espressioni generali per ottenere il K-esimo numero pari ed il K-esimo numero dispari. Completale con le opportune parti dichiarative.

VI) Scrivi uno statement iterativo che calcoli le somme dei primi N numeri dispari e dei primi N numeri pari. Completa il tutto con le opportune parti dichiarative. E' necessario usare in questo caso uno statement composto? Perchè?

VII) Uno studente vuole calcolare una tabella dei volumi delle sfere, in modo da non essere costretto a calcolarli ogni volta. Non vuole che il programma effettui il calcolo se gli viene fornito un raggio negativo, nè se il raggio della sfera è superiore al metro. La tabella che vuole fare è per raggi che variano di centimetro in centimetro, ed il volume deve venir espresso in m^3.
Scrivi gli statements dichiarativi opportuni e quindi gli statements iterativi che permettano di calcolare i volumi delle sfere solo quando sono soddisfatte le condizioni dette più sopra.

CAPITOLO 8

GLI STATEMENTS LOGICI

SCOPO DEL CAPITOLO

Alla fine di questo capitolo sarai in grado di

... sapere cosa sono ed a che servono gli statements condizionali in un programma

... sapere cos'è uno statement IF in PASCAL ed usarlo

... sapere cos'è uno statement CASE ed usarlo

... scrivere pezzi di programma in PASCAL con dichiarazioni, statements di assegnazione, statements ripetitivi, statements logici

... sapere cosa si intende per "programmazione strutturata"

... scrivere un esempio di programma seguendo le tecniche della programmazione strutturata.

GLI STATEMENTS CONDIZIONALI IN GENERALE

Un altro dei cardini di un linguaggio di programmazione è la possibilità offerta al programmatore di modificare il corso del programma a seconda delle particolari condizioni che si possono verificare o per informazioni esterne (per esempio per la battitura di un certo tasto della telescrivente) o per condizioni interne al programma stesso.

Un tipico esempio in questo campo è la semplice divisione. Supponi quindi che ad un certo punto del programma tu voglia eseguire la semplice istruzione

```
X := A/B;
```

con X, A e B di tipo REAL. E supponi anche che B non sia un numero su cui ha molto controllo, dal momento che è il risultato di calcoli precedenti, abbastanza complessi. Ebbene: sarà opportuno che tu stia molto attento ad eseguire la divisione stessa, dal momento che potrebbe capitare che B sia nullo, ed il calcolatore in tal caso si rifiuterebbe di eseguire una divisione per 0. Così pure una semplice radice quadrata può mettere in crisi un programma:

```
Y := SQRT (X);
```

non è eseguibile se X risulta minore di 0. E gli esempi si potrebbero moltiplicare.

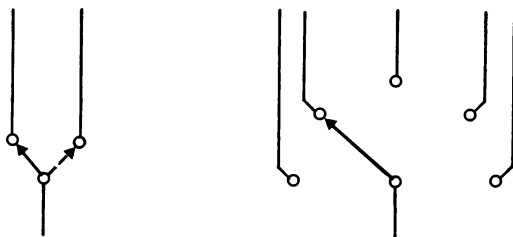
In tutti questi casi è essenziale quindi avere uno strumento che permetta di modificare il normale corso delle istruzioni a seconda delle condizioni che vengono a crearsi nel programma stesso. In altri termini il programma deve essere in grado di prendere delle **decisioni logiche**. Inutile dire che tutti i vari casi che si possono presentare devono venire previsti da chi scrive il programma stesso!

Il tutto finisce quindi per assomigliare molto ad un parco ferroviario, con un gioco di scambi che vengono attivati dagli stessi treni che passano.

Tornando all'esempio della radice quadrata, è necessario avere delle istruzioni che eseguano un compito di questo genere:

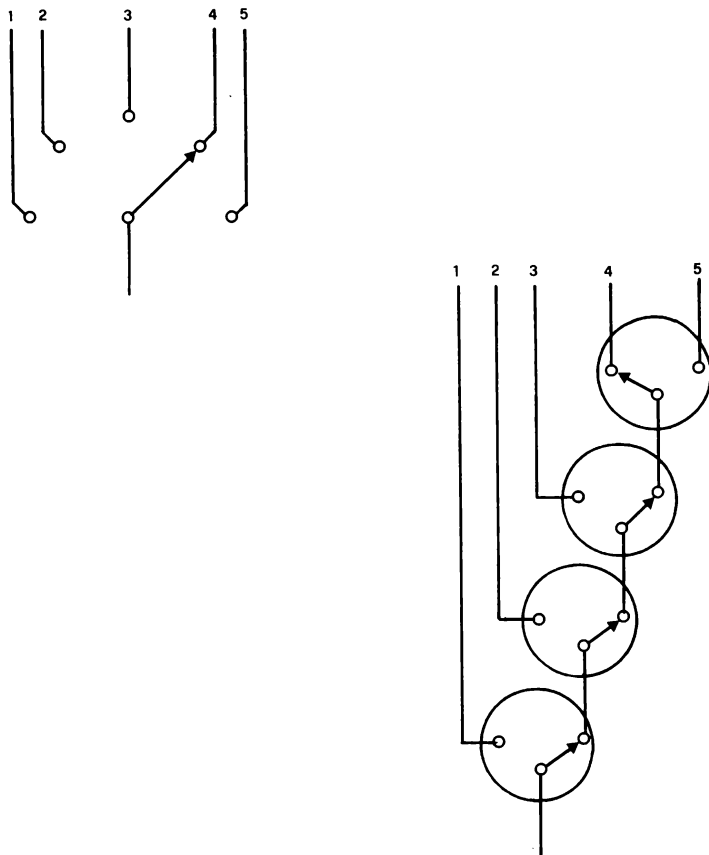
> "dopo aver calcolato la X esamina il suo valore. Se è maggiore od uguale a 0 esegui pure la radice quadrata e poi procedi. Se capita invece che sia minore di 0 dai una segnalazione di errore e quindi esci" (oppure "richiedi altri dati", oppure "fermati ed aspetta istruzioni", oppure. . .)

Uno statement di questo genere assomiglia molto ad un interruttore, e si chiama anche uno "statement condizionale", proprio perchè il flusso del programma dipende da delle **condizioni** che in esso vengono a verificarsi. In PASCAL uno statement di questo tipo è lo statement **IF** (che vuole dire "SE").

In un programma possono però verificarsi casi più complessi e decisioni più complicate da descrivere. Facciamo un esempio commerciale, e supponiamo che un albergo abbia una lista di nomi di clienti nella quale è incluso anche il numero di volte in cui questi signori sono venuti a villeggiare nell'albergo stesso, e supponiamo che l'albergatore voglia trattare meglio i clienti più affezionati. Per fare questo (magari per assegnare loro le camere migliori, o per spedire per tempo prima che agli altri le richieste di prenotazione) ha bisogno di liste differenziate: una lista per i clienti che vengono da più di 10 anni, una per coloro che vengono da 7 a 9 anni, una per coloro che vengono da 4 a 6 anni, una per coloro che si son fatti vedere due o tre volte, e l'ultima per coloro che si son fatti vedere una volta sola e che magari non si faranno vedere mai più visto che hanno — diciamo — **dimenticato** di saldare il conto. Ebbene: la lista dei clienti andrà esaminata ed il numero di volte in cui il cliente sarà apparso in albergo sarà la **chiave** (si chiama proprio così) di una decisione. Solo che questa volta la definizione non fra un semplice sì od un no. Anche se la faccenda è in fondo riducibile ad un gioco di sì-no successivi, sarebbe molto più comodo che il cliente venisse incasellato in un singolo colpo in una delle liste di cui sopra. Uno statement di questo genere assomiglia ad un interruttore a molte vie, cioè ad un commutatore secondo uno schema che possiamo vedere in figura

Nella prima parte della figura vediamo lo schema di una decisione che può far scorrere

il flusso del programma solo su due diverse strade. Questo è il caso della IF di cui parlavamo prima. Nella seconda parte c'è invece lo schema di un programma che invece è in grado di decidere fra molte strade diverse, proprio come succederebbe ad un contatto rotante che può far scorrere la corrente attraverso diversi rami del circuito, a seconda della posizione che occupa.

Lo statement che in PASCAL esegue un lavoro di questo genere si chiama **CASE** ("caso"), ed è uno statement estremamente sintetico e potente, che conviene imparare bene ed usare meglio.

Tanto per chiudere l'argomento vediamo nella figura successiva come un contatto rotante può benissimo essere realizzato con una serie di interruttori semplici. Possiamo concludere che uno statement come CASE non porta a nulla di nuovo dal punto di vista concettuale, quindi. Così come un commutatore non porta a nulla di nuovo, e potrebbe benissimo venir realizzato con una serie di interruttori. C'è solo il fatto che è molto più semplice ruotare una manopola di quanto non sia il far scattare (e controllare) la posizione molte levette.

Uno statement condizionale è quindi in grado di modificare il flusso del programma a seconda del verificarsi o meno di una condizione logica.

Data la generalità del linguaggio PASCAL a questo punto dovrebbe esserti già ovvio come si scriveranno le parti essenziali di uno statement di questo tipo.

I) occorrerà specificare la "condizione logica". Quindi occorrerà definire una variabile booleana; e questa sarà definita — ovviamente — in tutta generalità, con il possibile uso di tutti gli operatori logici che abbiamo imparato a conoscere

II) occorrerà poter definire due statements (eventualmente composti) che verranno eseguiti l'uno nel caso in cui la variabile sia vera, l'altro nel caso in cui sia falsa.

Questo per quanto riguarda la IF, cioè l'interruttore semplice.

Per quanto riguarda invece il "commutatore" occorrerà definire in qualche modo una variabile (non booleana, stavolta) che possa assumere diversi valori. È già da questo si può vedere che non sarà affatto necessario che tale variabile sia del tipo INTEGER: basterà che sia di tipo scalare qualunque. Dopo di che occorrerà definire i vari statements o gruppi di statements che devono venire eseguiti a seconda dei valori che la variabile di controllo può assumere. Ed occorrerà anche dire al programma quali sono questi valori in modo che esso possa prendere le decizioni opportune.

Col che abbiamo già — più o meno — definito la sintassi di questi statements. Si tratta ora di vedere in pratica come si fa.

LO STATEMENT IF

I tipi base di IF sono due, di cui ecco il BNF

⟨statement IF⟩ : : = IF ⟨espressione booleana⟩
THEN ⟨statement⟩ ;

⟨statement IF⟩ : : = IF ⟨espressione booleana⟩
THEN ⟨statement⟩ ELSE ⟨statement⟩ ;

Come vedi, questi due statements ricalcano esattamente ciò che avevamo detto nel paragrafo precedente.

Analizziamo le differenze fra i due.

Nel primo IF lo statement (che naturalmente può essere anche del tipo "composto", cioè lunghissimo e complicatissimo, purchè racchiuso nelle parole BEGIN . . . END) viene eseguito **solo se** la espressione booleana risulta **vera**. Altrimenti viene ignorato. Questo è il tipo più semplice di IF, e — detto per inciso — è quello di uso più frequente, dal momento che anche nei programmi la norma non è la complicazione esasperata, ma piuttosto un'onesta semplicità.

Nel secondo IF ci sono **due** statement: il primo viene eseguito nel caso in cui

l'espressione booleana sia vera. L'altro nel caso in cui questa sia falsa. Ti ricordo che THEN vuol dire "allora", ed ELSE "altrimenti".

Questo mi ricorda che abbiamo appena accennato alla cosiddetta "programmazione strutturata". In realtà molta gente ne parla e quando si leggono articoli in materia si ha l'impressione che sia una cosa in cui si perderebbero anche i più universali geni matematici passati, presenti e futuri. Nulla di più falso. In realtà la complicazione e l'astrusità degli articoli derivano in genere dal fatto che chi scrive non ha affatto le idee chiare nemmeno lui. La realtà della "programmazione strutturata" è molto più semplice. L'idea di base è che si dovrebbe evitare il più possibile l'uso di un linguaggio di programmazione, inteso come complesso sistema di regole da affidare alla memoria e da perfezionare con la pratica. L'ideale sarebbe poter descrivere il problema al calcolatore nel linguaggio di tutti i giorni; nella nostra lingua, insomma. Magari anche a voce e non per iscritto. In attesa che tutto ciò diventa realtà (e molti di noi pensano che lo diverrà nei prossimi 10-20 anni) ci si accontenta di descrivere al calcolatore il nostro problema in un linguaggio **simile** al nostro, ma **semplificato**, organizzato con un minimo di sintassi standard; o — se preferisci — **"strutturato"**. Un "inglese strutturato", quindi. Un po' un compromesso fra il nostro linguaggio libero e delle regole fisse che il calcolatore deve pur avere, visto che ancora non è capace di seguire tutte le sfumature del linguaggio umano.

Quindi, anzitutto l'uso di un "inglese strutturato". In altri termini: statements che assomiqlino molto al parlare comune. Ed a questo proposito sarà bene notare che — se vuoi programmare in PASCAL — la conoscenza dell'inglese ti tornerà estremamente utile. Pensa — ad esempio — a cosa diverrebbe la IF vista in italiano:

⟨nuova IF, cioè statement SE⟩ : : =
    SE ⟨espressione booleana⟩ ALLORA ⟨statement⟩
        ALTRIMENTI ⟨statement⟩ ;

e capirai da solo che un "italiano strutturato" vorrebbe dire "scrivere un programma senza pensarci su".

Questo è infatti lo scopo della programmazione strutturata: arrivare a mettere a punto un sistema di tecniche che vanno dal linguaggio sintatticamente corretto, alla scelta della sintassi negli statements, alla programmazione a blocchi e così via e che permettano a chi scrive un programma di **"parlare del"** suo problema. Quindi di definirlo sempre meglio, scrivendo su un pezzo di carta la descrizione a parole del problema stesso. Il problema stesso si precisa via via, e la soluzione si perfeziona sempre di più. Alla fine del processo ci si trova ad avere la soluzione scritta nei dettagli su un pezzo di carta. Ma — attenzione, qui sta il punto chiave! — se si è avuta l'accortezza di usare un inglese semplificato ("strutturato") e se si usa un linguaggio di programmazione "strutturato", a questo punto il lavoro è praticamente finito: la soluzione scritta sul pezzo di carta **è praticamente** il programma.

Devo dire che tutto ciò è molto spontaneo se la lingua madre di chi scrive è l'inglese. Meno per noi italiani. Possiamo consolarci col fatto che tutti i musicisti di tutto il mondo praticamente devono sapere l'italiano. Ed anche questa non è in fondo una piccola cosa.

Altra nota a margine di questa digressione: se mai il calcolatore sarà capace di riconoscere la voce umana (speriamo di sì, sarebbe bellissimo) una delle lingue più

facili da fargli riconoscere sarà proprio l'italiano per la sua chiarezza di vocali e consonanti e per la sua assenza (in pratica) di regole di pronuncia (si parla come si scrive). Non resta che stare a vedere. E darsi da dare nel frattempo.

Alla fine di questo capitolo vedremo in concreto come si imposta un problema con la programmazione strutturata e come si arriva — in modo indolore — a scrivere il programma relativo.

LO STATEMENT CASE

Ecco al solito il BNF dello statement

⟨statement CASE⟩ : : = CASE ⟨espressione **non reale**⟩ OF
    ⟨elemento della lista CASE⟩ ; ⟨elemento della lista CASE⟩ END;

⟨elemento della lista CASE⟩ : : =
    ⟨lista dei labels CASE⟩ : ⟨statement⟩
    | ⟨nulla⟩

⟨lista dei labels CASE⟩ : : = ⟨label CASE⟩ , ⟨label CASE⟩

⟨label CASE⟩ : : = ⟨costante⟩

A questo punto sarà bene fare degli esempi. Per complicarti la vita ho deciso di **non** dare degli esempi giusti, ma degli esempi che contengono degli errori. Il numero degli errori è dichiarato prima di ogni esempio. A te scoprirli, eventualmente ripassandoti la sintassi del linguaggio finora studiata.

GLI ESEMPI

I) una IF (un errore)

```
IF A<0 AND C3=C5 THEN
   BEGIN
     X := K +7 * (SQRT(B)–0.78); Y=X +5;
   END;
```

II) un'altra IF (un errore)

```
IF (L>9) OR (FF<>0 AND KK=0) THEN
   BEGIN;
     A := B–C; X := F OR G;
   END;
ELSE V := V * V;
```

III) Un CASE (un errore)

```
CASE K OF
   1: X :=3;
   2: L=L+1;
   3: Z :=SQRT(Y);
   4: BEGIN K :=3 K; Y :=2 END
END;
```

IV) Un altro CASE (un errore)

```
CASE K*(K–45) OF
3,3.7 : X :=X +2;
29,32: W :=Y +3;
–5,43, –3: BEGIN  H :=H +1;  Z :=SQRT(X)  END
END;
```

V) Un altro CASE. Supponiamo che la variabile CH sia definita del tipo CHAR. C'è un errore

```
CASE CH OF
'A': X :=–2;
'C', 'D', 'K':  BEGIN;  A :=3;  B :=A +2;  END;
'B', 'E':
END;
```

Nota che in questo caso se la variabile CH vale B o E non succede nulla: il programma semplicemente continua.

UNA PICCOLA NOTA SUL "NESTING"

Capita spesso di avere delle IF "una dentro l'altra", e questo porta a discutere un attimo questo caso dal punto di vista del tempo di esecuzione. Conviene infatti mettere nei livelli più esterni delle IF (cioè come IF che vengono eseguite **per prime**) quelle IF che da un'analisi del problema sono verificate **più di rado**. Se si fa così si ottiene il risultato che quasi sempre l'intero blocco di IF "salta", in quanto la prima IF non è verificata, e quindi è inutile procedere oltre.

Non sempre si può decidere a tavolino su queste questioni: ma in genere è tempo bene impiegato, che può far risparmiare delle impressionanti quantità di tempo al calcolatore, rendendo il programma, in fase di esecuzione, molto più veloce, oltrechè facilmente controllabile.

Facciamo un esempio, e supponiamo di avere un campione di popolazione che viene descritto tramite età, statura e sesso. Vogliamo conoscere le stature delle donne che superano i 80 anni. Poichè sappiamo a priori che — se il campione di cui stiamo parlando non è selezionato in base a qualche criterio particolare — donne e uomini sono presenti nella popolazione in percentuali all'incirca uguali, mentre le persone di più di 80 anni non sono molte nella popolazione, sarà saggior far fare al programma un ragionamento del genere

I) la persona (uomo o donna che sia) ha più di 80 anni?

II) se non li ha procedi ad esaminare la prossima. E questo sarà il caso più frequente, che permetterà al programma di cavarsela molto spesso con un solo confronto

III) se li ha chiediti se è uomo o donna

IV) se è uomo non interessa, e procedi alla prossima persona.
Anche in questo caso il programma avrà "sprecato del lavoro", e per scartare gli elementi che non interessano stavolta avrà avuto bisogno di **due** confronti. Però le persone scartate con **due** confronti saranno molto poche (persone con più di 80 anni e per di più **uomini**).

V) se è donna procedi coi calcoli che vuoi fare. Quando hai finito riprendi da capo.

Come risultato avremo quindi che il programma impiegherà poco tempo a selezionare il campione interessante, visto che di solito scarterà le persone non interessanti ai fini del problema con un solo confronto.

Se avessimo invertito l'ordine dei confronti ci saremmo dovuti chiedere anzitutto se la persona era uomo o donna. Questo ci avrebbe permesso di scartare il 50% circa della popolazione (cioè gli uomini) con un solo confronto, e poi avremmo avuto bisogno di due confronti per scartare dall'elaborazione ulteriore tutte le donne al di sotto degli 80 anni. Il risultato, ai fini della selezione del campione sarebbe lo stesso, ma il tempo impiegato dal programma per giungere a tanto sarebbe più lungo proprio perchè il programma sarebbe stato costretto a fare più elaborazioni per arrivare allo stesso risultato

Vale la pena quindi in questi casi di spendere un po' di tempo per analizzare il problema, mettendo come primi i confronti su condizioni che si verificano più di rado.

UN ESEMPIO DI PROGRAMMAZIONE STRUTTURATA

Prenderemo come esempio le solite equazioni di II grado. E scriviamo tutti i ragionamenti in inglese, per abituarci ad usare la programmazione strutturata.

Anzitutto ti ricordo la formula risolutiva:

se $D = B^2 - 4AC$ è $\geqslant 0$ ci sono due radici reali date da

$$X_1 = \frac{-B - \sqrt{D}}{2A} \qquad X_2 = \frac{-B + \sqrt{D}}{2A}$$

se $D = B^2 - 4AC$ è < 0 ci sono due radici complesse coniugate date da

parte reale $-\frac{B}{2A}$

parte immaginaria $\frac{\sqrt{-D}}{2A}$

Ed ecco il procedimento, passo passo, in inglese

I) **VAR**iables needed

A, B, C : coefficients of equations. Type **REAL**
X1, X1 : solutions of the equation. Type **REAL**
D : discriminant of equation. Type **REAL**
F : Flag. Type **CHAR**. Wille be 'R' real solutions, 'C' for complex solutions

II) Compute discriminant

$D = B^2 - 4AC$

III) **IF** D $\geqslant$ 0 **THEN**

a) put F = 'R'
b) compute $X1 = (-B-\sqrt{D})/(2A)$
c) compute $X2 = (-B+\sqrt{D})/(2A)$
d) exit

one statement only!

ELSE

a) put F = 'C'
b) compute real part $X1 = -\frac{B}{2A}$
c) compute imaginary part
$X2 = \frac{\sqrt{-D}}{2A}$
d) exit

one statement only!

IV) **END** of the program

Non penso sia difficile da seguire. Ora possiamo riprendere in mano questa descrizione schematica e scrivere direttamente il programma in PASCAL. Segui il procedimento (istruzione per istruzione) e verifica che in pratica si tratta semplicemente di mettere in forma un po' più schematica quanto già scritto più sopra. In altri termini: il programma **è già scritto.**

```
VAR A, B, C, D, X1, X2:  REAL;  F:CHAR;
BEGIN
   D := B * B - 4 * A * C;
   IF D >=0 THEN
      BEGIN
         F :='R';  X1 := (-B-SQRT(D))/(2 * A):
                   X2 := (-B+SQRT(D))/(2 * A)
      END;
   ELSE                        (
      BEGIN
         F :='C';  X1 := -B/ (2 * A);
                   X2 :=SQRT(-D)/(2 * A)
      END;
END.
```

RIASSUNTO DI QUESTO CAPITOLO

⟨statement IF⟩ : : = IF ⟨espressione booleana⟩ THEN ⟨statement⟩ ;

⟨statement IF⟩ : : = IF ⟨espressione booleana⟩
THEN ⟨statement⟩ ELSE ⟨statement⟩ ;

⟨statement CASE⟩ : : = CASE ⟨espressione **non** reale⟩ OF
⟨elemento della lista CASE⟩ ; ⟨elemento della lista CASE⟩ END;

⟨elemento della lista CASE⟩ : : =
⟨lista dei labels CASE⟩ : ⟨statement⟩ | ⟨nulla⟩

⟨lista dei labels CASE⟩ : : = ⟨label CASE⟩ , ⟨label CASE⟩

⟨label CASE⟩ : : = ⟨costante⟩

PROBLEMI ED ESEMPI

I) Riprendi il problema VII del capitolo precedente e risolvilo con l'uso delle IF

II) Riprendi il problema III) del capitolo precedente e risolvilo con l'uso delle IF

III) Scrivi il BNF delle IF e spiegalo con parole tue

IV) Scrivi il BNF dello statement CASE e spiegane l'uso con parole tue

V) Scrivi — completa di parti dichiarative — la parte centrale di un programma che restituisca, dato il mese dell'anno, il numero di giorni in esso contenuto

VI) Completa il punto V) supponendo che il programma abbia anche conoscenza dell'anno in cui siamo, e decida di conseguenza se l'anno è bisestile o no..

VII) Supponi di sapere l'anno in cui siamo, e che giorno della settimana è il capodanno. Scrivi la parte centrale di un programma destinato a fornire il giorno della settimana una volta che gli venga dato il mese ed il giorno del mese stesso.

VIII) Scrivi la parte centrale di un programma che dati dei numeri fra 1 e 10 costruisca delle stringhe di caratteri uguali ai numeri romani corrispondenti.

IX) Scrivi la parte centrale di un programma che sia in grado di trasformare un angolo dato in gradi, primi e secondi in gradi decimi e centesimi di grado. Completa il tutto con le rispettive parti dichiarative.

X) Esistono in PASCAL alcune "funzioni standard" che fanno parte del sistema stesso. Alcune le abbiamo già viste (ORED, ORD. .). Un'altra di queste è la RANDOM che ogni volta che viene chiamata restituisce un numero di tipo REAL, compreso fra 0 ed 1. Così scrivere

```
X := X + RANDOM;
```

fa sì che alla X, qualunque essa sia, venga aggiunto un numero a caso, compreso fra 0 e 1.
Premesso questo scrivi la parte centrale di un programma in cui si aggiungono successivamente ad una variabile X intera dei numeri a caso **compresi fra –10 e 10** ed interi anche loro. Inserisci anche queste limitazioni:

a) se il numero che devi aggiungere è pari lascialo come sta se invece è dispari, cambiagli il segno

b) se la somma diviene minore di –10 o maggiore di 10 riportala nell'intervallo –10. . .10 aggiungendo o togliendolo, rispettivamente.

Programmi di questo genere sono spesso usati per generare piacevoli giochi grafici su un display, magari a colori.

CAPITOLO 9

I DATI STRUTTURATI-GENERALITA'

SCOPO DEL CAPITOLO

Alla fine di questo capitolo sarai in grado di

... sapere cosa si intende per operazione di un programma

... sapere cosa si intende per dato di un programma

... sapere cosa si intende per dato scalare

... sapere cosa si intende per dato strutturato

... sapere perchè è necessario spesso usare dei dati strutturati

... sapere perchè i dati strutturati devono essere di generi diversi per poter avere maggiore flessibilità nel loro trattamento

IL PROGRAMMA

Avevamo detto che un programma si presenta come una serie di **azioni** che vengono effettuate su dei **dati.** Poichè abbiamo in pratica finito gli statements di PASCAL possiamo già tirare le somme: le azioni sono in effetti descritte da degli statements, e poco importa se questi sono delle espressioni aritmetiche, o delle iterazioni complicate, o degli statements condizionali. In realtà si tratta sempre di modificare o dei dati o il corso del programma stesso **usando** ed **agendo su** dei dati. Nota che per **dato** non necessariamente devi intendere un numero, in PASCAL. Può essere un valore booleano, o un carattere, o una variabile di tipo nuovo, definito da te.

Quindi noi abbiamo già visto quali sono le azioni elementari che PASCAL può compiere. E non vorrei che a questo punto tu pensassi che tutto sommato queste sono abbastanza poche, perchè ti assicuro che coi pochi mattoni elementari che abbiamo visto ed appreso a conoscere si possono costruire programmi di incredibile complessità. Dai problemi gestionali, alla più raffinata matematica moderna. Basti dire che PASCAL offre molto più del FORTRAN, e che proprio sul FORTRAN l'intera comunità scientifica mondiale è vissuta (e vive ancora, fra l'altro: il FORTRAN non morirà così presto) per un ventennio abbondante.

Quindi le azioni di PASCAL possono essere raggruppate così

I) espressioni (in senso generale)

II) iterazioni

III) decisioni logiche

Ed in questi gruppi è compreso tutto lo spettro di azione di questo linguaggio.

In realtà un programma non consiste solo di **azioni**, ma di **azioni che operano su dei dati.** E qui sta il grosso del discorso: cosa sono stati infatti per noi finora i dati di PASCAL? Diciamolo pure: ben poca cosa. In pratica o dei numeri, o dei caratteri, o dei valori booleani, o dei tipi extra. Comunque in generale dei "tipi scalari".

Vediamo meglio cosa intendiamo con questa frase nel prossimo paragrafo.

I TIPI SCALARI

Anche di ciò abbiamo già parlato. Con questa frase in PASCAL intendiamo una cosa molto semplice: i dati, che possono per loro definizione stessa assumere diversi valori (nel senso generalizzato che abbiamo visto), sono individuati dal programma singolarmente, uno per uno.

Facciamo un esempio. La statura di un individuo può essere espressa da un numero reale, cui associeremo un identificatore che magari chiameremo proprio STATURA. Così l'età, in anni, potrà essere anche lei un numero, e potremo decidere noi se usare il tipo intero o reale. Supponiamo di usare anche in questo caso il tipo REAL, per ragioni che vedremo più avanti, e di chiamarlo con l'identificatore ETA (non sono permessi accenti negli identificatori).

Fin qui tutto chiaro: ogni volta che diremo STATURA il programma andrà a prendere in una particolare locazione della memoria **un certo numero**, e su quello potrà operare con le azioni (statements) che noi avremo programmato. E così con ETA o con altre variabili che ci potrebbero servire nel nostro immaginario programma.

Facciamo un passo più in là. E supponiamo stavolta di **non** avere un singolo individuo, ma **tre** persone. Delle quali vogliamo statura ed età. Cominciamo ad essere nei guai. In realtà dovremo inventarci tre identificatori per la statura e tre per l'età.

Finora niente di drammatico: potremo benissimo inventarci i seguenti identificatori

STATURA1	STATURA2	STATURA3
ETA1	ETA2	ETA3

ed il problemino sarebbe risolto.

Per esempio la media delle stature sarebbe facile calcolarla:

```
MEDIA := (STATURA1 + STATURA2 + STATURA 3)/3;
```

e puoi pensare da te stesso ad altri esempi.

I guai cominciano quando le persone cominciano ad aumentare. Che succederebbe se volessimo avere la media delle stature di 10 persone? Un primo guaio è dovuto al fatto che gli identificatori in PASCAL sono distinguibili dai loro primi **8** caratteri. Ed in questo caso STATURA1 e STATURA10 vengono presi come identificatori identici! Anche qui poco male. Potremmo cambiare il nome dell'identificatore così

S1 S2 S3 S4 S5 S6 S7 S8 S9 S10

Però se dovessimo fare la media di 10 stature lo statement relativo comincia a diventare impressionante

MEDIA := (S1 + S2 + S3 + S4 + S5 + S6 + S7 + S8 + S9 + S10)/10;

E le persone sono solamente 10. Che succederebbe se diventassero 100, o 1000 o 5000? Succederebbe una cosa molto semplice: saremmo veramente nei guai, dal momento che operazioni molto semplici (quali la media) diverrebbero estremamente complesse da descrivere. Ben presto si raggiungerebbe l'impossibilità **pratica** di effettuarle.

E' possibile ovviare a questo inconveniente? In altri termini: è possibile avere un formalismo che ci permetta di maneggiare in modo semplice ed agevole grosse masse di dati?

I DATI STRUTTURATI

Non hanno niente a che vedere con la programmazione **strutturata**, anzitutto. Se vuoi diciamo pure che il termine non è dei più felici.

Riprendiamo l'esempio delle medie, del paragrafo precedente, e vediamo qual'è la causa delle nostre difficoltà.

Anzitutto un fatto: i dati sono **molti**. Se fossero pochi non avremmo problemi di sorta. Ma non è tanto importante che siano molti i **dati**. Il guaio vero è che non sono molti gli **identificatori**. E questo a sua volta è dovuta al fatto che noi pretendiamo di usare **un** identificatore per **ogni** dato. Cioè di indicare i dati singolarmente, uno per uno, con nomi diversi. E' a questo punto che non abbiamo scampo: all'aumentare dei dati da gestire aumenterà di pari passo il numero degli identificatori e si allungherà la codifica del programma.

Quindi abbiamo anche trovato una possibile soluzione al nostro problema:

> dobbiamo trovare il modo di dare a dei dati che giudichiamo simili per qualche ragione (alla statura per esempio) un nome unico, collettivo, cui il programma possa fare riferimento. Nel contempo dobbiamo dare al programma la possibilità di riferirsi al nome collettivo, e — specificato questo — di individuare un dato **singolo**. Magari la statura dell'individuo che occupa il posto 3863 nella nostra immaginaria lista delle stature.

Risolto questo problema il programma non avrebbe più bisogno di riferirsi singolarmente ad ogni dato, e le difficoltà di cui sopra sparirebbero.

Qualcosa di simile esiste anche nella vita pratica. Guai se si pretendesse di indicare singolarmente tutti i volumi contenuti nelle biblioteche italiane! In realtà si procede in modo diverso: per indicare **il singolo volume** si va per passi successivi

I) si indica la biblioteca in cui si trova

II) poi della biblioteca si indica la stanza

III) poi nella stanza si indica lo scaffale

IV) poi nello scaffale si indica il posto che il volume vi occupa

Ed a questo punto non ci sono più limiti: possiamo anche individuare una singola lettera fra **tutte** quelle contenute in **tutti** i libri di **tutte** le biblioteche italiane: basta che aggiungiamo altri passi

V) indichiamo la pagina in cui si trova la lettera

VI) indichiamo la riga in cui si trova la lettera

VII) indichiamo il posto che la lettera occupa nella riga

L'esempio è volutamente paradossale. A noi non interessa affatto andare a prendere **una** lettera fra tutte quelle che esistono in tutte le biblioteche italiane. Ma è interessante il fatto che proprio perchè le lettere sono organizzate in righe, pagine volumi, scaffali, stanze, biblioteche noi possiamo con relativa facilità riferirci ad una sola di esse, compiendo sette soli passi di scelta successivi. Prova a pensare (per confronto) a cosa succederebbe se tutte le lettere fosseo individuate da nomi **diversi**.

Rendere possibile quindi il maneggio di grosse quantità di dati passa obbligatoriamente attraverso una fase — per così dire — organizzativa. Occorre raggruppare i dati in **strutture**, che eventualmente possano poi essere combinate fra loro per ottenere strutture via via più complesse. E' il classico procedimento di "nesting" anche questo: tornano fuori sempre le bambolotte russe o le scatole cinesi. Ed è importante che i singoli dati siano riconoscibili uno per uno nell'ambito di una struttura. Se poi una serie di strutture è chiamata con un nome diverso, e costituisce così una **struttura di strutture**, il concetto deve rimanere valido.

Così con le lettere (il dato scalare), si può combinare una riga (struttura di livello 1). Ed in una riga una lettera è facile trovarla. Con le righe (**strutture**, ormai, non più **dati scalari!**) si può combinare una pagina (struttura di livello 2 o struttura di strutture), ed anche in questo caso non è drammatico trovare una riga in una pagina. E con le pagine si può fare un libro (struttura di strutture di strutture, o strutture di livello 3). Ed ancora non è certo complicato trovare la pagina di un libro.

Il risultato è quindi che con i dati strutturati viene reso possibile maneggiare grosse quantità di dati, potendosi riferire al singolo dato, senza dover usare tanti nomi diversi quanti sono i dati da esaminare.

I TIPI STRUTTURATI IN PASCAL

Al di là degli statements, nei quali PASCAL presenta novità eleganza, concisione,

ma in fondo nulla di straordinario, la vera potenza del linguaggio si manifesta nell'importanza data alla organizzazione dei dati. Forse è questo che ha fatto pensare a PASCAL — ai suoi inizi — come ad un linguaggio squisitamente devoluto a problemi gestionali. Non è assolutamente così. Proprio perchè lo uso ti posso assicurare che nel campo scientifico è estremamente valido, consentendo una compattezza, una facilità ed una chiarezza che finora sono a mio parere assolutamente ineguagliate.

L'importanza dei dati strutturati in PASCAL si misura anche dai diversi **tipi di dati strutturati**. Mentre in FORTRAN ce n'è solo uno (la matrice o array) in PASCAL ce ne sono ben quattro (ARRAY, SET, RECORD, FILE), più un altro, che non si sa proprio se è un tipo o no (il POINTER), viste le sue caratteristiche estremamente speciali. E non è esagerato dire che in un programma serio e di una certa complessità una considerevole parte dello sforzo di chi costruisce il programma va spesa nel decidere a quale tipo di struttura appoggiarsi per fare migliore uso delle possibilità offerte dal linguaggio.

Anche in questo caso non esistono ricette di alcun genere. E' un mestiere che ha una sola ricetta: il **fare**. E a questo proposito è bene che ti dica subito una cosa: non illuderti che dopo aver letto questo libro saprai programmare in PASCAL. Questo succederà solo se oltre a leggere e a eseguire i problemi avrai la possibilità di trovare i problemi stessi su un vero calcolatore che abbia l'uso del PASCAL. Purtroppo (o forse no!) le materie scientifico-tecniche si imparano solo facendo: leggere anche bene libri anche ben scritti è solo il primo passo.

CAPITOLO 10

IL TIPO ARRAY

SCOPO DEL CAPITOLO

Alla fine di questo capitolo sarai in grado di

... sapere che cosa si intende per ARRAY

... sapere come si dichiara l'uso di un ARRAY in un programma

... sapere come si identifica un elemento di un array

... sapere come viene immagazzinato in memoria un array

... scrivere dei programmi che contemplino l'uso degli arrays, oltre alle istruzioni che finora abbiamo imparato ad usare.

IL TIPO ARRAY

Questo è il primo tipo di dati strutturati che esamineremo. La ragione di ciò non è affatto o logica o propedeutica. Diciamo che da un lato è un po' storica (questo è in fondo il primo di dati strutturati apparso nei linguaggi di programmazione) e da un altro un po' casuale. In altri termini: bisogna pur incominciare da qualcosa.

Il tipo ARRAY è dunque una struttura di dati, la quale per potersi chiamare ARRAY deve soddisfare ad alcune condizioni:

I) gli elementi che compongono la struttura devono essere tutti dello stesso tipo, detto anche tipo base o tipo dei componenti (base type, component type)

II) il numero dei componenti deve essere fissato all'inizio del programma e non può più essere alterato

III) il punto I) è cambiato, nelle ultime versioni di PASCAL sotto l'onda della sollevazione popolare. Pertanto nei PASCAL più avanzati il numero dei componenti può essere lasciato libero, e fissato di volta in volta nel corso del programma. Vedremo caso per caso l'utilità di tutto ciò.

Dunque: un numero fisso di componenti (al massimo modificabile da programma), ma comunque dello stesso tipo. O REAL, o INTEGER, o CHAR, o BOOLEAN o "di fantasia".

A tutto questo insieme di componenti viene attribuito un nome unico, collettivo: appunto il nome dell'array. E con ciò viene soddisfatto il primo requisito stabilito per i dati strutturati, quello cioè di avere un meccanismo per dare un nome collettivo a molti dati singoli messi assieme.

Come possiamo individuare **un singolo** elemento dell'array? Nella maniera più semplice: l'array è una struttura ordinata e quindi basta dire — per individuarne un elemento — il posto che esso occupa nell'array stesso. Detto in altri termini: l'informazione del posto che un elemento occupa nell'array è specificata in una variabile, che è detta **indice** (index) e che di solito è di tipo INTEGER ma non necessariamente. Specificando il posto che l'elemento occupa nell'array (e quindi il valore di questa variabile) noi siamo in grado di individuare l'elemento senza possibilità di errore.

Un array non va confuso con la più familiare "matrice", che parecchi di voi conosceranno. Se mai ha parentela con una "matrice ad un solo indice", che di solito viene anche detta "vettore", presentandosi proprio come un'estensione dei vettori che abbiamo studiato alle suole medie. Va detto però che un array è qualcosa di ancor più generale: una matrice — sia pure ad un solo indice — è pur sempre un insieme di numeri **reali** che vengono individuati da un indice **intero**. Qui invece siamo in presenza di elementi di qualunque tipo individuati da indici che possono essere anche questi di qualunque tipo, eccezion fatta per il tipo reale. Anche questa è una restrizione ovvia, dal momento che coi numeri nel tipo REAL non si può **contare**, nel senso usuale del termine.

Un array può essere pensato come una pila di piatti. Per individuare un piatto in una pila di piatti rossi, dovremo anzitutto dare un nome alla pila (ROSSA, evidentemente) e poi dire in che posto si trova il piatto nella pila ROSSA.

Ma con gli arrays si possono costruire strutture più complesse: col classico procedimento di "nesting" si possono fare degli arrays i cui elementi siano anche essi degli arrays. In questo caso per individuare un elemento dovremo aggiungere un passo in più:

I) dovremo specificare il nome dell'array di arrays

II) dovremo specificare il posto che in questa struttura occupa l'array che contiene l'elemento che cerchiamo. Questo posto verrà individuato da una variabile che si chiamerà "primo indice", o "indice di primo livello".

III) individuato l'array che contiene l'elemento in questione dovremo individuarlo dicendo che posto occupa in questo array. Questa informazione sarà data da una seconda variabile, che si chiamerà "secondo indice" o "indice di secondo livello".

In altri termini, alla fine di questo complicato procedimento concettuale ci troviamo fra le mani una ricetta piuttosto semplice: per individuare un elemento basta dare il nome dell'array e poi specificare non **uno** ma **due** indici. E naturalmente il procedimento può venir continuato, con arrays ancora più complessi.

Tornando all'esempio delle pile di piatti, supponiamo ora di avere oltre alla ROSSA altre tre pile di piatti, rispettivamente (e **nell'ordine** ricordati che l'array deve: invece di essere una struttura **ordinata**: se così non fosse non potrebbe bastare un solo numero

ad individuare un elemento) BIANCA, VERDE, AZZURRA. E supponiamo di fare sulla tavola di casa nostra una "pila di pile di piatti", mettendo prima la pila AZZURRA, poi la VERDE sopra a questa, poi la BIANCA, poi la ROSSA. Otterremo una nuova pila. In quest'unica pila potremo individuare un piatto specificando

a) di che colore è. Il che equivale a dire a quale delle pile componenti appartiene. Il primo indice in questo caso **non** sarebbe di tipo INTEGER, come vedi

b) che posto occupa nella pila prescelta. Questo secondo indice sarebbe, invece di tipo INTEGER.

Val giusto la pena di notare che non è affatto necessario che il numero di piatti che compongono le singole pile sia sempre lo stesso. Le cose funzionano ugualmente bene anche se esso varia da pila a pila.

Ed infine una nota a proposito degli indici. Per ragioni storiche varie, oltre che di praticità nella compilazione dei programmi da parte dei calcolatori è invalso l'uso di mettere gli indici (nel caso in cui un array ne abbia più di uno, cioè sia un array di arrays di arrays di. . .) in un ordine prestabilito. Precisamente l'ordine in cui vengono specificati gli indici è esattamente l'opposto di quanto siamo venuti finora dicendo. In altri termini: il calcolatore pretende che per individuare un piatto gli si dica, nell'ordine.

"il terzo piatto della pila verde"

e non

"prendi nella pila verde il terzo piatto"

Non sottovalutare questa piccola particolarità, perchè di solito è fonte di sofferenze. E' bene averla presente fin dall'inizio ed evitarsi futuri guai.

Incidentalmente questa convenzione vale per tutti i calcolatori: può essere sintetizzata in questo modo

"gli indici di un array vengono ordinati in modo che appaiano **prima** quegli indici che — scorrendo sugli elementi dell'array — variano **più rapidamente".**

Nelle matrici (di cui parleremo più avanti essendo un caso particolare degli array,) diremmo che gli elementi sono "memorizzati per colonne", e non "per righe".

LA SINTASSI

Ecco quindi il BNF delle dichiarazioni di ARRAY

⟨tipo base⟩ : : = ⟨tipo semplice⟩ | ⟨tipo strutturato⟩
| ⟨tipo pointer⟩

⟨tipo semplice⟩ : : = ⟨tipo scalare⟩ | ⟨tipo subrange⟩
| ⟨identificatore di tipo⟩

⟨tipo ARRAY⟩ : : = ⟨ARRAY [⟨tipo dell'indice⟩ {, ⟨tipo dell'indice⟩}
OF ⟨tipo dei componenti⟩ ;

oppure

⟨tipo ARRAY⟩ : : = ARRAY [⟨tipo dell'indice⟩]
{OF ARRAY [⟨tipo dell'indice⟩]}
OR ⟨tipo dei componenti⟩ ;

⟨tipo dell'indice⟩ : . = ⟨tipo semplice⟩

Ci sono quindi due possibilità per dichiarare un ARRAY: la prima dichiarando subito quali sono gli indici che si useranno; la seconda dichiarando l'array proprio come una "pila di pile di piatti". E' inutile farti notare che la prima forma è di gran lunga la più semplice ed immediata, e quindi la più usata, fino al punto che la seconda forma è praticamente caduta in disuso e nemmeno viene più introdotta nei RASCAL più moderni.

Facciamo ora degli esempi. Supponiamo di dover definire una matrice di numeri reali definita a due indici: il primo che varia da 1 a 4 ed il secondo che varia da 2 a 5. Ecco come dovremo scrivere la dichiarazione:

VAR MATRICE1 : [1. .4, 2. .5] OF REAL;

ed a questo proposito ti ricordo che il tipo dell'indice può benissimo essere un "subrange". Siccome questo è un caso molto frequente sarà bene che tu ti riveda i tipi "subrange" nel caso tu non sia assolutamente sicuro di te stesso.

Per coloro che a scuola hanno fatto le matrici, sarà bene mettere in forma esplicita la matrice di cui alla dichiarazione precedente. Tanto per non complicare le cose gli elementi di questa matrice verranno indicati con "a", oltre agli indici, ovviamente. Ecco quindi come si presenta la matrice

$$\begin{pmatrix} a_{12} & a_{13} & a_{14} & a_{15} \\ a_{22} & a_{23} & a_{24} & a_{25} \\ a_{32} & a_{33} & a_{34} & a_{35} \\ a_{42} & a_{43} & a_{44} & a_{45} \end{pmatrix}$$

Mentre nella memoria del calcolatore questi numeri verranno memorizzati nel seguente ordine

$$a_{12} \quad a_{22} \quad a_{32} \quad a_{42} \quad a_{13} \quad a_{23} \quad a_{33} \quad a_{43} \quad a_{14} \quad a_{24} \quad a_{34} \quad a_{44} \quad a_{15} \quad a_{25} \quad a_{35} \quad a_{45}$$

ovvero seguendo l'ordine "per colonne" e non quello "per righe" che noi — leggendo — siamo portati spontaneamente a seguire. Indubbiamente il tipo subrange è uno dei tipi più usati per gli indici di un array. Ecco qui un altro esempio

VAR ARRAY2 : ARRAY [1. .7, –3. .8, –70. .–60] OF BOOLEAN;

in questo caso abbiamo un array a tre indici, i cui elementi possono essere falsi o veri. Nota che ho chiamato questo array col nome di ARRAY2. Questo è un permesso, dal momento che ARRAY è una parola riservata, ma ARRAY2 (costituito da soli 6 caratteri) viene capito dal calcolatore come un identificatore diverso di ARRAY.

Ed ecco come si può usare un indice di tipo diverso dal solito intero. Supponiamo

di definire un tipo MESE con la dichiarazione seguente:

TYPE MESE = (GEN, FEB, MAR, APR, MAG, GIU, LUG, AGO, SEP, OTT, NOV, DIC);

Potremo costruire un array in cui l'indice sia di questo tipo ed i cui elementi siano di tipo booleano. A che ci può servire? Magari a sapere se in un mese il riscaldamento è acceso o no. Per esempio

VAR RISCALD: ARRAY [MESE] OF BOOLEAN;

Negli elementi di questo array verranno quindi messi dei valori "vero" o "falso" a seconda che il riscaldamento nel mese in esame sia o no acceso.

GLI ELEMENTI DI UN ARRAY

Vediamo ora come, dato un array si possa indicarne un singolo elemento. Il procedimento è estremamente semplice. Al nome dell'array fanno seguito gli indici che individuano l'elemento in esame, nel loro ordine corretto e racchiusi in parentesi quadre, oltrechè separati da virgole.

Così ecco degli esempi per individuare gli elementi degli arrays precedentemente definiti:

MATRICE1	[2,4]	che sarà di tipo REAL
ARRAY2	[2,5,–65]	che sarà di tipo BOOLEAN
RISCALD	[OTT]	che sarà di tipo BOOLEAN

Inoltre non necessariamente gli indici devono essere delle costanti: potrebbero benissimo essere delle espressioni, purchè – s'intende – il tipo finale del risultato del calcolo delle espressioni sia del tipo previsto per l'indice.

Così per MATRICE1 potremmo avere qualcosa di questo tipo:

MATRICE1 [K +5, L * L –M]-

dove supporremo -- evidentemente – che K, L M siano del tipo INTEGER.

E' lecita adesso una domanda: cosa succede se le espressioni che abbiamo messo come indici danno come risultato dei valori che escono dai limiti imposti agli indici? per esempio: se nell'esempio precedente trovassimo

MATRICE 1 [200, 4378] ?

La risposta è semplice: il programma dà una segnalazione di errore e si blocca. Il che vuol dire che chi programma deve stare molto attento a che ciò non succeda; e spesso questa non è una impresa semplice. Del resto tutte le comodità si pagano sempre con qualche fatica in più.

LE OPERAZIONI DI PACKING E UNPACKING

Queste non sono operazioni che si possono spiegare in dettaglio, dal momento che

spesso la loro implementazione dipende dal sistema che si sta usando. In ogni caso qualcosa si può dire.

Facciamo il caso (che poi è quello più frequentemente usato) di un array di caratteri: ciò vuol dire in pratica che ogni elemento dell'array conterrà un carattere.

Supponiamo ora di avere una macchina che usi 16 bits per parola. Ciò vuol dire che ogni elemento dell'array occuperà 16 bits (due bytes). Sappiamo però, dalle codifiche standard, siano esse EBCDIC, ASCII o altre meno usate, che per definire un carattere basta un byte soltanto. Il risultato? che ogni parola che contiene un elemento del nostro ARRAY conterrà **un** byte di informazione **utile**, mentre l'altro byte sarà semplicemente **vuoto**, cioè sprecato.

In parole povere: procedendo in questo modo noi usiamo il doppio dello spazio che sarebbe necessario realmente per contenere la nostra informazione.

Ma le cose possono essere ancora più serie. Se fossimo in presenza di un array di variabili booleane ci troveremmo veramente di fronte ad una situazione paradossale: mentre basta **un** bit per una variabile booleana (che può assumere solo due valori) qui ci troveremmo ad usarne ben 16: i 16 bits dell'intera parola. Lo spreco di memoria a questo punto è notevole.

La risposta in PASCAL a questo problema è quella di usare degli ARRAY "impaccati" o "packed" che permettono proprio di risparmiare memoria quanto più possibile.

Vediamo il caso più frequente, che è quello di un array consistente di soli caratteri, ma "impaccato". In questo caso vengono messi **due** caratteri per singola parola (uno per byte) e l'array si presenta come una sequela ininterrotta di bytes tutti significativi e non sprecati. Una simile struttura prende anche il nome di "stringa" (string). La definizione di tipo esiste, anzi questo è un tipo standard di PASCAL. Se per esempio volessimo chiamare TESTO una simile struttura dovremmo dichiarare qualcosa come

VAR TESTO : PACKED ARRAY [1. .65] OF CHAR;

con ciò diremmo che in TESTO noi possiamo mettere fino a 65 caratteri, ognuno dei quali occuperà un byte di memoria, per cui comunque vadano le cose sarà il sistema a far sì che si occupino 65 bytes, al massimo arrotondando l'occupazione di memoria a seconda del numero di bytes di cui è fatta una parola. Nel caso di un 16 bits avremmo dunque un'occupazione totale di 66 bytes, cioè di 33 parole, contro le 65 che avremmo avuto se non avessimo usato questa possibilità.

Più difficile dire cosa succede per i PACKED ARRAYS di variabili di tipo diverso da CHAR. Qui andiamo sui dettagli del sistema che viene usato, e siccome non c'è ancora uno standard di PASCAL universalmente riconosciuto, preferisco rimandarti al manuale d'uso del tuo calcolatore, che ti darà in dettaglio ciò che puoi fare in questi casi.

Come si può fare a rivolgersi ad un singolo elemento di un array impaccato? Qui sorge un certo problema: il fatto è che in una sola parola del calcolatore trovano posto, proprio per effetto dell'impaccaggio parecchi elementi dell'array. Questo

non è molto grave nel caso in cui vengano impaccati dei caratteri, dal momento che il byte (in cui un carattere va sempre a finire) può sempre venir indirizzato singolarmente e singolarmente chiamato dal calcolatore. Nè per gli array di variabili booleane, qualora il calcolatore permetta di indirizzare il singolo bit senza complicazioni eccessive. Il problema sorge per tutti quei casi intermedi, in cui magari vengono impaccate in una parola informazioni che richiedono due o quattro bits. A questo punto per estrarne una i casi sono due: o si aggredisce il problema a livello di linguaggio macchina (cosa che si cerca sempre di evitare fino a che è possibile) o si usano delle funzioni standard che PASCAL offre all'utente. Queste sono le funzioni PACK ed UNPACK ed anche se non abbiamo ancora parlato di funzioni puoi facilmente immaginare cosa fanno. Supponiamo che GRANDE e PICCOLA siano due Arrays, il primo "normale" (e quindi "grande") ed il secondo "impaccato" (e quindi "piccolo").

In tal caso la funzione

PACK (GRANDE, K, PICCOLO)

permette di impaccare tanti elementi di GRANDE a partire da quello col posto K **incluso** in PICCOLO quanti ce ne possono stare. Al contrario la funzione

UNPACK (PICCOLO, GRANDE, K)

metterà in GRANDE tutti gli elementi che si possono ottenere "disimpaccando" gli elementi di PICCOLO a partire dal posto K incluso.

Le funzioni sono evidentemente l'una l'inverso dell'altra. Vale la pena di notare che non sempre sono disponibili in tutti i PASCAL commerciali. In realtà ciò non è molto strano: sono funzioni molto dispendiose in tempo e memoria, e la tendenza attuale è di sostituirle cogliendone lo spirito, ma evitando l'uso di queste possibilità comode più di nome che di fatto.

RIASSUNTO DI QUESTO CAPITOLO

⟨tipo base⟩ : : = ⟨tipo semplice⟩ | ⟨tipo strutturato⟩ | ⟨tipo pointer⟩

⟨tipo semplice⟩ : : = ⟨tipo scalare⟩ | ⟨tipo subrange⟩
| ⟨identificatore di tipo⟩

⟨tipo ARRAY⟩ : : = ARRAY [⟨tipo dell'indice⟩ {, ⟨tipo dell'indice⟩}]
OF ⟨tipo dei componenti⟩ ;

⟨tipo ARRAY⟩ : : = ARRAY [⟨tipo dell'indice⟩]
{OF ARRAY [⟨tipo dell'indice⟩]} OF ⟨tipo dei componenti⟩ ;

⟨tipo dell'indice⟩ : : = ⟨tipo semplice⟩

PROBLEMI ED ESEMPI

I) Spiega con parole tue cosa si intende per ARRAY in PASCAL

II) Scrivi il BNF delle dichiarazioni di ARRAY e fai degli esempi concreti di dichiarazioni

III) Con riferimento al punto II) scrivi un elemento degli array che hai dichiarato

IV) I numeri che sono racchiusi in parentesi quadre e servono per identificare gli elementi degli arrays si chiamano anche "sottoscritti" (subscripts) dall'uso corrente in matematica di scriverli in basso: così:

$$a_{1,3,7} \quad K_{6,9} \quad \text{etc.}$$

e gli arrays prendono spesso anche il nome di "variabili sottoscritte" (subscripted variables). Gli indici sottoscritti possono essere delle espressioni? Possono essere di tipo REAL? E perchè pensi sia giusta questa tua risposta?

V) Supponi di avere un array K di 100 numeri di tipo REAL. Scrivi la parte centrale di un programma che esamini tutti questi numeri e restituisca alla fine il più piccolo ed il più grande di essi. Programmi di questo genere sono usatissimi e vengono di solito costruiti come funzioni o sottoprogrammi a parte, avendo largo uso soprattutto quando si tratti di costruire dei programmi per la costruzione di grafici che debbono — ovviamente — essere opportunamente ridotti o allargati di scala in modo da rientrare in uno schermo od in un foglio.

VI) La media di un insieme di numeri viene definita come la loro somma, divisa per il loro numero. Supponi di avere dei numeri di tipo REAL in un array di 100 elementi. Scrivi tutte le parti dichiarative che ti servono e poi scrivi la parte del programma che calcoli la media dei numeri che stanno nell'array.

VII) Riprendi il problema II) del capitolo VII) (quello sui fattoriali) e calcola

$$1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} \ldots$$

continuando a procedere fino a che i termini che si aggiungono sono troppo piccoli per essere significativi. Ti puoi arrestare quando il termine da aggiungere è più piccoli di 10^{-8}. Non abbiamo fatto ancora pratica su input-output: altrimenti a questo punto potresti stampare il risultato e vedere che hai ottenuto un'eccellente approssimazione del numero "e", base dei logarismi naturali (o neperiani) e che vale 2.71828183. . . . .

VIII) Supponi di avere un array dei primi 1000 numeri interi. Il cosiddetto "crivello di Eratostene" permette di ottenere in modo abbastanza semplice i numeri primi ricorrendo ad un sistema drastico ma efficace: comincia col primo numero (due alla partenza), lascialo stare e cancella tutti i suoi multipli. Poi trova il numero successivo, non cancellato: lascialo stare, ma cancella tutti i suoi multipli. E procedi così finchè non ci sono più numeri a disposizione. Incidentalmente è per questa ragione che i numeri "primi" si chiamano così: proprio perchè sono i "primi" numeri che appaiono nel crivello, e gli unici a non venir cancellati.
Sapendo questi, scrivi la parte utile di un programma, completo di dichiarazioni, che partendo da un array che contenga i primi 1000 numeri interi, applichi il metodo di Eratostene ed esca con i numeri primi. Puoi usare vari metodi per "cancellare" un numero che viene stacciato dal crivello: metterlo uguale a

0 o cambiargli segno, o altro, basta che poi nella logica del crivello che tu userai tenga conto di questi fatti.
Alla fine riscrivi i soli numeri primi in un array a parte, e contali.

IX) Nei moderni PASCAL è permesso avere un array di dimensioni variabili, per cui ci possono essere delle dichiarazioni di tipo

ARRAY GIANNI [1. .N] OF REAL;

e solo al momento dell'esecuzione N verrà specificato (magari fornendolo direttamente al calcolatore da tastiera) e l'array GIANNI acquisterà delle dimensioni "vere".
Sapendo questo generalizza il programma precedente al caso in cui l'array di partenza è di dimensioni arbitrarie N.

X) Il crivello di Eratostene è un eccellente esercizio di programmazione, avendo in sè stesso molta logica che può essere anche resa molto raffinata. Per esempio: saresti capace di scrivere un programma per il crivello senza usare gli arrays?

XI) Si definisce come prodotto di due matrici il cosiddetto "prodotto righe per colonne", che penso possa meglio venir spiegato con un esempio:

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{pmatrix} \times \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \\ B_{31} & B_{32} \end{pmatrix} =$$

$$\begin{pmatrix} A_{11}\ B_{11} + A_{12}\ B_{21} + A_{13}\ B_{31} & A_{11}\ B_{12} + A_{12}\ B_{22} + A_{13}\ B_{32} \\ A_{21}\ B_{11} + A_{22}\ B_{21} + A_{23}\ B_{31} & A_{21}\ B_{12} + A_{22}\ B_{22} + A_{23}\ B_{32} \end{pmatrix}$$

e puoi vedere che gli elementi della matrice prodotto, che potremmo scrivere anche così

$$\begin{pmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{pmatrix}$$

si ottengono moltiplicando gli elementi di una riga per gli elementi di una colonna (e facendone poi la somma) delle matrici fattori. Puoi anche vedere da te come si scelgono le righe e le colonne nelle matrici fattori per ottenere la matrice prodotto, ed anche il fatto che il numero delle colonne della prima matrice deve essere uguale al numero delle righe della seconda.
Sapendo questo scrivi un programma completo che calcoli il prodotto di due matrici generiche di dimensioni arbitrarie, ferme restando le condizioni prima dette perchè se ne possa effettuare il prodotto.
Alla fine controlla accuratamente che nel caso particolare di questo esempio tu ottenga effettivamente i risultati previsti.

XII) Nello spazio i vettori vengono descritti da **tre** numeri reali (le loro tre componenti). Quindi puoi descrivere un vettore come un array di tre elementi.
Ricordandoti le formule del calcolo vettoriale scrivi dei programmi completi di intestazione e dichiarazioni che effettuino le operazioni che sai si possono fare tra i vettori. Cioè una prima "biblioteca di algebra vettoriale". Controlla in qualche caso particolarmente facile i tuoi programmi e commentali in dettaglio..

CAPITOLO 11

IL TIPO RECORD

SCOPO DEL CAPITOLO

Alla fine di questo capitolo sarai in grado di

... sapere cos'è un RECORD in PASCAL

... sapere come si dichiara un tipo RECORD

... sapere cos'è un campo ed un identificatore di campo

... scrivere la dichiarazione di un record semplice

... sapere cosa si intende per "variante" di un record

... scrivere una struttura ad albero di un record con varianti

... scrivere in PASCAL la dichiarazione di un tipo RECORD con varianti

... sapere cosa si intende per "statement WITH"

... usare lo statement WITH

IL TIPO RECORD

Sarà bene — per capire cos'è un record — esaminarne le principali differenze con l'array

I) Un array ha un numero fisso di **elementi** mentre un record ha un numero fisso di **campi.**

II) non necessariamente la struttura del record è fissa: ce ne possono essere diverse, che vengono chiamate "varianti"

III) a differenza dell'array un record può contenere dati di qualunque tipo. All'interno di un singolo campo però un solo tipo è ammesso.

Un record è in effetti la più generale struttura dei dati offerta da PASCAL. In esso si può mettere praticamente di tutto.

Così un record può contenere dati di inventario. Di ogni singolo oggetto può contenere il numero di inventario, una descrizione dell'oggetto magari a parole, la data

d'acquisto, dati su fatture e bolle di consegna. Poi se è in uso o in magazzino. E se è in magazzino dove si trova; mentre se è in uso, chi lo usa, se ha subito manutenzioni e/o riparazioni e da quando: chi, dove e perchè le ha fatte, quanto sono costate. . .

Inutile dire cosa può diventare un record per i dati anagrafici. Proprio la sua versatilità e la possibilità di contenere delle varianti a seconda dei valori di alcuni campi consente un'estrema flessibilità e versatilità nella costruzione di un archivio anagrafico. Così un record può contenere nome, cognome, luogo e data di nascita di una persona poi il codice fiscale ed il suo stato civile, e — a seconda dei vari casi che si possono a questo punto presentare — tutti i dati inerenti al matrimonio (se è sposato), o al matrimonio ed al decesso del coniuge (se vedovo/a), ed eventualmente dati sui matrimoni successivi, etc. etc. In realtà il limite è veramente la fantasia.

LA SINTASSI DELLA DICHIARAZIONE

Al solito ecco il BNF della dichiarazione di RECORD

⟨tipo RECORD⟩ : : = RECORD ⟨lista di campi⟩ END;

⟨lista di campi⟩ : : = ⟨parte fissa⟩
| ⟨parte fissa⟩ ; ⟨parte variabile⟩
| ⟨parte variabile⟩

⟨parte fissa⟩ : : = ⟨sezione del record⟩ {; ⟨sezione del record⟩}

⟨sezione del record⟩ : : = ⟨identificatore del campo⟩
{, ⟨identificatore del campo⟩} : ⟨tipo⟩ | ⟨nulla⟩

Vedi dal BNF che un RECORD è costituito da delle entità che vengono chiamate **campi**, e che questi oltre a possedere un nome cui si può in futuro far riferimento sono identificati da un **tipo**. E' bene a questo punto non dimenticare che i tipi non sono solamente quelli scalari, subrange o di fantasia, ma anche quelli di dati strutturati. Così un campo di un record potrebbe benissimo essere un array, o un record a sua volta. In realtà esistono delle limitazioni a quest'altro gioco di scatole cinesi, e le limitazioni sono date dal calcolatore che si ha a disposizione. Non esistono regole fisse, ma è bene in ogni caso informarsi se delle strutture di strutture possibili sulla carta sono o no permesse nel particolare sistema che si ha a disposizione.

L'analogia che abbiamo fatto nel capitolo scorso per l'array, come insieme di tutte le lettere che compongono una biblioteca, può essere estesa adesso al record.

Potremo assimilare un record ad un insieme di oggetti disparati che sono messi su degli scaffali, i quali stanno in delle stanze con delle porte aperte su un corridoio. Non sempre tutte le porte sono aperte (cioè non tutte le varianti del record sono attivate). Quelle che lo sono danno però accesso a delle stanze (i campi del record) ognuna delle quali contiene gli oggetti più disparati. Senza contare il fatto che potrebbe darsi benissimo che una di queste porte dia accesso non ad una stanza con oggetti, ma ad un altro corridoio sul quale si aprono delle porte che portano ad altre stanze, e così via.

Capirai da te che questa è una struttura di dati estremamente versatile e flessibile,

dal momento che permette di usare in una struttura unica un enorme numero e di elementi e di tipi diversi. Non è esagerato dire che chi sa usare **bene** il record in PASCAL si può già considerare un esperto di questo linguaggio.

UN ESEMPIO SEMPLICE

Spero che chi legge queste righe conosca almeno per sentito dire i numeri complessi, dal momento che non possiamo spiegare anche questo in un testo che parla di un linguaggio di programmazione.

Brevemente possiamo ricordare che i numeri complessi sono definiti da una **coppia** di numeri reali. Normalmente vengono scritti come

$$x + i\,y$$

x prende il nome di **parte reale**, y il nome di **parte immaginaria** ed i è l'**unità immaginaria** che soddisfa alla proprietà $i^2 = -1$.

Possiamo usare i numeri complessi in PASCAL? Nota che in FORTRAN il numero complesso è un tipo standard, così come il REAL, l'INTEGER, il BOOLEAN, il CHAR. In PASCAL ciò non è ancora successo, e pare che delle Società Segrete stiano preparando delle dimostrazioni di piazza per costringere gli estensori dei prossimi PASCAL a creare anche il tipo COMPLEX come tipo standard. Anche qui non resta che aspettare.

In attesa che ciò avvenga (qualcosa è già avvenuto, a proposito: esiste un tipo standard nuovo, la STRING di cui parleremo più avanti, che è straordinariamente comodo ed usatissimo) non ci resta che dire come si fa ad insegnare a PASCAL ad usare i numeri complessi.

Da quanto ti ho detto dovrebbe risultare chiaro che un numero complesso è in realtà una struttura di dati. Molto elementare, è vero, ed è per questo che l'ho scelta come primo esempio. Si tratta di una struttura che consiste di **due numeri REAL.**

Potremo quindi costruire un RECORD che consiste di due campi, entrambi di tipo REAL. Questi due campi verranno chiamati RE e IM (abbreviazioni di parte REale e parte IMmaginaria), ed ecco come verrà fatta la dichiarazione di tipo, che ti consiglio di controllare col BNF, per sicurezza:

```
TYPE COMPLEX = RECORD RE,IM: REAL
                      END;
```

Da questa dichiarazione in poi noi avremo a disposizione un nuovo tipo da usare per variabili e costanti, e dichiarazioni di questo genere

```
VAR W,Z,RLC : COMPLEX;
```

saranno perfettamente lecite.

Ci resta però da vedere come si fa ad identificare un elemento del singolo record (che si chiama "componente"). La sintassi è estremamente semplice:

⟨nome del componente⟩ : : = ⟨nome del record⟩ . ⟨identificatore del campo⟩

Il che in pratica vuol dire che se W è una variabile definita COMPLEX con W.RE e W.IM potremo indicare la sua parte reale e la sua parte immaginaria rispettivamente. E così potremo anche operare coi numeri complessi. Se per esempio W, Z e T sono numeri complessi e vogliamo che sia

T = W + Z

non potremo fare certo questa somma così come la effettuiamo fra dei numeri reali. La matematica ci insegna che la parte reale di T sarà la somma delle parti reali di W e Z. E che così succederà pure per le parti immaginarie. Per ottenere la somma dovremo scrivere dunque **due** istruzioni:

```
T.RE :=W.RE + Z.RE;  T.IM :=W.IM +Z.IM;
```

ed a questo punto ci sarà in T la somma dei due numeri complessi di partenza.

Più complicato il caso del prodotto, che in questo caso diverrebbe

```
T.RE :=W.RE * Z.RE – W.IM * Z.IM;
T.IM :=W.IM * Z.RE +W.RE * Z.IM;
```

e lascio a te controllare che gli statements e le formule da cui essi derivano sono effettivamente corretti.

UN ESEMPIO NORMALE

Possiamo definire un tipo DATA che consista del giorno della settimana (tipo da definire), del giorno del mese (che è un intero ma limitato a valori che vanno da 1 a 31), del mese (che è anche questo un tipo da definire, e dall'anno (che è un intero).

I campi del nostro record saranno duqnue SETT (per "giorno della settimana"), GIORNO (per "giorno del mese"), MESE ed ANNO.

Ecco quindi una possibile dichiarazione:

```
TYPE DATA = RECORD SETT   : (LUN, MAR, MERC, GIO, VEN, SAB, DOM);
                   GIORNO : 1. .31;
                   MESE   : (GEN, FEB, MAR, APR, MAG, GIU, LUG,
                            AGO, SEP, OTT, NOV, DIC);
                   ANNO   : INTEGER;
                   END;
```

A questo punto saranno lecite dichiarazioni del tipo

```
VAR D1, D2: DATA;
```

e — per esempio — D1.ANNO sarà un INTEGER, che conterrà l'anno della data nel record D1. E così D2.GIORNO sarà un altro intero, che sarà limitato nel range 1. .31.

UN ESEMPIO COMPLETO E LE VARIANTI

Un record può avere una struttura non fissa, ma variabile da record a record, e possono variare sia il numero che il tipo dei componenti. In ogni caso devono essere dichiarate tutte le varianti possibili fin dall'inizio, cioè dalla dichiarazione del record, e ricadono in quella parte del BNF che all'inizio abbiamo chiamato "parte variabile" e che adesso è opportuno esaminare in più dettaglio. Ecco quindi il BNF dettagliato:

⟨parte variabile⟩ : : = CASE ⟨identificatore di campo⟩ :
  ⟨identificatore di tipo⟩ OF ⟨variante⟩ {; ⟨variante⟩}

⟨variante⟩ : : = ⟨lista di labels CASE⟩ : (⟨lista di campo⟩)
  ⟨nulla⟩

⟨lista di labels CASE⟩ : : = ⟨label CASE⟩ {, ⟨label CASE⟩}

⟨label CASE⟩ : : = ⟨costante⟩

⟨identificatore di campo⟩ : : = ⟨identificatore⟩ : | ⟨nulla⟩

Vediamo adesso un caso concreto, e supponiamo di voler organizzare i dati concernenti la situazione di una biblioteca. Sarà bene in ogni caso schematizzare prima ciò che vogliamo e solo in un secondo momento pensare alla stesura in PASCAL.

Le informazioni su un libro potranno quindi essere contenute in un record che possiamo schematizzare così

a) record per la descrizione della situazione di un libro

- I) Autore (nome e cognome: 20 + 20 lettere)
- II) Titolo (40 lettere)
- III) Data dell'edizione (mese ed anno)
- IV) Casa Editrice (20 lettere)
- V) Luogo di edizione (Città e Nazione: 40 lettere)
- VI) Data di edizione (mese ed anno solamente)
- VII) Sigla del catalogo (10 fra lettere e numeri)
- VIII) Esistenza in biblioteca. Vari casi possibili:
  - a) se presente (booleano)
  - b) se prestato (nome, cognome, indirizzo, data dell'ultimo prestito, numero dei prestiti)
  - c) se dal rilegatore (nome, cognome, indirizzo, data)
  - d) se distrutto o perso (data dell'ultima presenza in biblioteca).

Come vedi si tratta di una struttura piuttosto complessa (e lo potrebbe essere di più). Un cosiddetto "schema ad albero" aiuta molto a seguire la situazione:

PA 1-20 sta per PACKED ARRAY [1. .20] Detto anche tipo STRING
DATA1 e DATA2: Tipi da definire
DATA1: Mese (Tipo a parte), Anno (INTEGER)
DATA2: Giorno (INTEGER), Mese (Tipo a parte), Anno (INTEGER)
Presenza in biblioteca: Tipo da definire

Uno schema ad albero è essenziale quando si tratta di progettare la struttura di un record. Tieni presente che un record può essere una struttura estremamente complessa ed articolata, e che gli esempi che stiamo facendo qui sono relativamente semplici. Nel caso di strutture più complesse può riuscire veramente difficile seguire tutti i rivoli e rivoletti in cui un record può spezzarsi. Non è male aiutarsi allora con degli schemi: quello che ti ho mostrato per la biblioteca è solo un esempio. Non è affatto escluso che tu trovi qualche schema molto migliore di questo.

Passiamo allora alla codifica in PASCAL: anzitutto avremo bisogno di alcuni tipi che ci renderanno la vita più facile in seguito: eccoli

```
TYPE   NOME =PACKED ARRAY [1. .20] OF CHAR;
       STRINGA =PACKED ARRAY [1. .40] OF CHAR;
       GIORNO =1. .31;
       MESE = (GEN, FEB, MAR, APR, MAG, GIU, LUG, AGO, SEPT, OTT,
              NOV, DIC);
       PRESENZA = (PRES, RILEG, PRESTATO, ASSENTE);
```

Definiti così i tipi che ci serviranno in seguito possiamo passare a definire i tipi più complessi, cioè le date: e ti faccio notare che di queste ne abbiamo **due**: uno che contiene solo il mese e l'anno, e l'altro che contiene giorno, mese ed anno. Ecco quindi le definizioni

```
TYPE DATA1 = RECORD
             MESE1  :  MESE;
             ANNO1  :  INTEGER;
             END;
TYPE DATA2 = RECORD
             DI2    :  GIORNO;
             MESE2  :  MESE;
             ANNO2  :  INTEGER;
             END;
```

A questo punto sono definiti tutti i tipi che ci serviranno in seguito e possiamo cominciare a stendere la definizione del record principale, che chiameremo — evidente mente — LIBRO

```
TYPE LIBRO = RECORD
             NOMAUT, COGAUT  :  NOME;
             TITOLO          :  STRINGA;
             ED1DATA         :  DATA1;
             EDITORE         :  NOME;
             LUOGOED         :  STRINGA;
             DATAED          :  DATA1;
             CATALOGO        :  SIGLA;
             SITUAZ          :  PRESENZA;

CASE SITUAZ  :  PRESENZA OF
  PRES   :  (DISPON  :  BOOLEAN)
  RILEG  :  (RNOME, RCOGN, RVIA  :  NOME; RNUM,RCAP: INTEGER;
            RCITTA: NOME; RDATA: DATA2);
```

```
    PRESTATO : (PNOME, PCOGN, PVIA: NOME; PNUM, PCAP: INTEGER;
                PCITTA: NOME; PDATA: DATA2; PTOT: INTEGER);
    ASSENTE  : (ASS: BOOLEAN; ASDATA: DATA2);
END,
```

Come vedi non è che sia una faccenda semplicissima, però bisogna riconoscere che è il problema in sè che non è certo semplice, e che se uno vuole avere tutto questo bagaglio di informazioni entro il calcolatore, occorrerà pure che gli dica cosa vuole!

Ci sono alcuni rilievi da fare che è bene non dimenticarsi

I) può darsi che per modifiche al programma od altro capiti ad un certo punto che nella lista dei CASE un campo dichiarato con un LABEL risulti vuoto. A meno di non cambiare tutta la struttura del record è bene ricordare che anche in questo caso esso va dichiarato. Magari con un semplice paio di parentesi vuote. Così se supponiamo che non ci interessa, da un certo momento in poi, l'informazione contenuta nel campo ASSENTE (magari perchè abbiamo trovato un filantropo che paga i libri mancanti di tasca sua) potremo evitare di mettere questa informazione, però il campo andrà comunque dichiarato

ASSENTE . ( );

a meno di non cambiare **anche** il tipo PRESENZA, facendo sparire la voce ASSENTE. In altri termini: nella parte delle varianti, controllata da CASE **tutte** le varianti possibili **devono** essere esplicitamente dichiarate.

II) Tutti i nomi dei campi devono essere **distinti**. Attenzione: **anche se appartengono a varianti diverse.** Questa è una condizione estremamente importante, ed è anche una comune causa di errori.

III) In una lista di campi è permesso l'uso di **una sola variante**, che deve seguire la parte fissa. E' però permesso il nesting di varianti. In altri termini, nel nostro esempio **non** avremmo potuto mettere un altro CASE di seguito al primo. Questa è considerata una limitazione inevitabile, dal momento che se non si facesse così si assisterebbe ben presto ad una crescita esponenziale di varianti, in certi problemi, ed il controllo della struttura uscirebbe facilmente dalle capacità di qualsiasi calcolatore presente e futuro. Tanto più che è permesso il **nesting di varianti:** il che permette la scrittura di records di estrema articolazione.
Per esempio, avremmo potuto fare una variante nella variante, dicendo che se il libro è stato prestato ad un adulto tutto resta come già detto, mentre se viene prestato ad un bambino occorre aggiungere anche il nome, cognome ed indirizzo di chi si prende la responsabilità per lui, o magari il nome della scuola in cui il bambino studia.

Una nota a margine, PASCAL ha una straordinaria versatilità e varietà nell'organizzazione dei dati. Ci si accorge ben presto, usandolo, che occorre almeno altrettanto sforzo, inventiva, mestiere per organizzare i dati di quanto non ne serva per scrivere il programma vero e proprio. Per di più nessuno può dirvi come dovete fare: così come non esistono ricette per risolvere un problema nel modo più semplice, più corto o più elegante, non esistono nemmeno ricette per organizzare i dati nella maniera più chiara compatta e concisa. Questi sono veramente mestieri che si imparano solo scrivendo programmi. I libri servono unicamente a dare una prima base: il resto viene solo con la pratica.

COME SI INDIVIDUANO GLI ELEMENTI DI UN RECORD

Gli elementi di un record possono essere chiamati ripercorrendo il cammino della definizione dell'informazione.

Così

LIBRO.RNOME e LIBRO.RCOGN

danno di ritorno il nome ed il cognome del rilegatore (sempre nel caso che il volume sia in legatoria, altrimenti ritorna un messaggio che informa l'utente dell'inesistenza nel record di questi campi).

Il numero dei prestiti che il libro ha subito si chiamerà LIBRO.PTOT ed il numero di catalogo LIBRO.CATALOGO. L'anno in cui è avvenuto l'ultimo prestito si chiamerà. LIBRO.PDATA.ANNO e sarà — ovviamente — di tipo INTEGER. E così via.

Quindi in generale possiamo dire che per individuare uno dei componenti del record si scrivono di seguito i nomi dei vari campi cui l'elemento appartiene, separati fra loro da un punto. Va da sè che conviene usare dei nomi e degli identificatori descrittivi il più possibile, ma anche il più possibile brevi, per evitare di dover scrivere troppo al momento di dover identificare un componente del record. Ancora una volta bisogna riconoscere che l'inglese tecnico, con la sua possibilità di abbreviazioni, le sue parole corte e la sua totale assenza di accenti è una lingua estremamente adatta questo genere di lavoro. L'italiano — purtroppo — è da sconsigliare decisamente, proprio per le ragioni già dette.

LO STATEMENT WITH

E' abbastanza chiaro da quanto si è detto che per individuare l'elemento di un record le regole sono semplici, la cosa è fattibile (per non dire intuitivamente semplice), però c'è un difetto: per identificare l'elemento di un record occorre scrivere molto.

Ecco un esempio: per individuare il mese in cui il libro è stato portato dal rilegatore (posto che ci si sia accertati che effettivamente è in legatoria e non è stato nè prestato, nè perduto o distrutto) occorre scrivere

LIBRO.RDATA.MESE

e le cose potrebbero essere ancora più complicate se si avessero identificatori più lunghi, o maggiori dettagli nel record.

PASCAL offre una scorciatoia a chi si trova in questa situazione. Si tratta dello statement WITH, che adesso esamineremo.

Anzitutto la sintassi:

⟨statement WITH⟩ : : = <u>WITH</u> ⟨variabile di record⟩
{, ⟨variabile di record⟩} <u>DO</u> ⟨statement⟩

Vediamo prima a cosa serve questo statement e poi vedremo dei casi concreti.

Riprendiamo dunque l'esempio del nostro record LIBRO. Se dovessimo scrivere un programma che lavora con gli elementi del record, ci troveremmo ogni volta a dover specificare cose del tipo

LIBRO.TITOLO   LIBRO.EDITORE   LIBRO.RDATA......

tutte frasi che hanno in comune — evidentemente — "LIBRO." all'inizio. Proprio basandosi su questa faccenda viene spontanea l'idea di sottintendere "LIBRO." e di lavorare col resto. E' appunto questa la funzione di WITH.

In questo caso tutta la parte di programma che usa gli elementi del record potrebbe essere compresa in una parentesi logica BEGIN ... END e diventare quindi un unico statement per quanto riguarda PASCAL. In secondo luogo si potrebbe far precedere questo statement composto da un WITH così congegnato:

```
WITH LIBRO DO
   BEGIN
      ............
   END
```

e ciò significherebbe per il programma che tutti gli identificatori dei campi del record possono venir semplificati, dal momento che per essi verrebbe appunto sottintesa la parte "LIBRO." comune a tutti. Così, con lo statement WITH, per indicare il titolo del libro basterebbe scrivere TITOLO, ed il programma — come effetto dello statement WITH e — capirà che intendiamo scrivere LIBRO.TITOLO, e così via.

Se poi vogliamo lavorare con gli elementi della data in cui il libro è stato portato in legatoria, possiamo evitarci di scrivere

LIBRO.RDATA.DI2   LIBRO.RDATA.MESE2   LIBRO.RDATA.ANNO2

scrivendo uno statement WITH di questo tipo

WITH LIBRO, RDATA DO

e da questo momento in poi gli elementi del record di cui sopra potranno essere semplicemente scritti

DI2   MESE2   ANNO2

Vedi quindi che lo statement WITH ti permette di scrivere molto meno, sostanzialmente ricorrendo al fatto che si possono sottintendere le parti comuni agli identificatori dei campi. I programmi ne guadagnano in leggibilità e chiarezza, cosa che è uno degli scopi fondamentali di PASCAL.

Gli statements WITH possono essere soggetti a nesting ad ogni livello: in altri termini, entro lo statement che viene dopo la parola DO possono trovare posto altre WITH. Così potremmo avere situazioni di questo tipo:

```
WITH LIBRO DO
BEGIN
   ...
   WITH RDATA DO
   BEGIN
   ...
   END;
END:
```

ed analoghe: in casi come questi possiamo scrivere solamente la parte più significativa dell'identificatore del campo, evitando sistematicamente di scrivere ogni volta tutte quelle specifiche che possono essere sottintese in un intero gruppo di statements.

RIASSUNTO DI QUESTO CAPITOLO

⟨tipo RECORD⟩ : : = RECORD ⟨lista dei campi⟩ END ;

⟨lista dei campi⟩ : : = ⟨parte fissa⟩
| ⟨parte fissa⟩ ; ⟨parte variabile⟩ | ⟨parte variabile⟩

⟨parte fissa⟩ : : = ⟨sezione del record⟩ {; ⟨sezione del record⟩}

⟨sezione del record⟩ : : = ⟨identificatore del campo⟩
{, ⟨identificatore del campo⟩} : ⟨tipo⟩ | ⟨nulla⟩

⟨parte variabile⟩ : : = CASE ⟨identificatore del campo⟩
⟨identificatore di tipo⟩ OF ⟨variante⟩ {; ⟨variante⟩}

⟨variante⟩ : : = ⟨lista di labels CASE⟩ : (⟨lista di campo⟩)
| ⟨nulla⟩

⟨lista di labels CASE⟩ : : = ⟨label CASE⟩ {, ⟨label CASE⟩}

⟨label CASE⟩ : : = ⟨costante⟩

⟨identificatore di campo⟩ : : = ⟨identificatore⟩ : | ⟨nulla⟩

⟨statement WITH⟩ : : = WITH ⟨variabile di record⟩
{, ⟨variabile di record⟩} DO ⟨statement⟩

PROBLEMI ED ESEMPI

I) Spiega con le tue parole, possibilmente per iscritto, le differenze fra ARRAY e RECORD

II) Scrivi il BNF del tipo RECORD e spiega il significato delle varie metavariabili usate

III) Spiega cosa sono i campi di un record e a cosa servono le varianti del record.

IV) **Senza guardare il testo** scrivi la dichiarazione di un record per la data (giorno della settimana, giorno del mese, mese, anno). Aggiungi una variante in cui ci sia l'informazione che dice se il giorno in questione è giorno lavorativo o no. Puoi usare una semplice variabile di tipo BOOLEAN, oppure una variabile di un tipo inventato da te. Alla fine confronta con quanto riportato nel testo e controlla che il tutto funzioni.

V) Se sei pratico di numeri complessi, scrivi la dichiarazione di un numero complesso che venga descritto in modulo e fase, e non tramite parte reale e parte immaginaria.

VI) Scrivi la dichiarazione di un record per dati anagrafici. Questo dovrebbe contenere
   a) nome, cognome e sesso
   b) luogo, provincia, nazione e data di nascita
   c) indirizzo di residenza (completo di CAP)
   d) numero di telefono (**se** la persona ha il telefono)
   e) codice fiscale e numero della carta d'identità

   Attenzione ad usare dati e campi realistici!

VII) Con riferimento al problema precedente: come faresti a scrivere la parte utile di un ipotetico programma che volesse selezionare le persone nate in un certo anno? Come faresti a selezionare le persone che **non** hanno il telefono? Scrivi le parti essenziali di questi programmi in PASCAL.

VIII) Se hai pratica di fatture commerciali, prova a scrivere la dichiarazione di un record che contenga le informazioni che ritieni utili per identificare una fattura (chi l'ha emessa, dove sta, il numero di codice e la data della fattura, informazioni sul materiale cui si riferisce, se è stata pagata, o è in corso di pagamento, o non è ancora in pagamento, o è contestata, o . . .).

IX) Con riferimento al problema VII) in che cosa lo statement WITH ti semplificherebbe il programma?

X) Descrivi con le tue parole a cosa serve lo statement WITH nell'identificazione di un campo di un record.

XI) Scrivi la dichiarazione di un record per le targhe automobilistiche. Tieni presente in modo attento il sistema di numerazione delle targhe, che cambia quando è stato raggiunto il milione e poi cambia ancora quando la prima lettera della targa ha raggiunto la lettera Z. Tieni presente che per ragioni di leggibilità non tutte le lettere dell'alfabeto sono usate. Puoi per il numero di targa usare semplicemente il tipo INTEGER? Perchè? Come puoi rappresentare un numero di targa, misto di lettere e di numeri?

CAPITOLO 12

IL TIPO SET

SCOPO DEL CAPITOLO

Alla fine di questo capitolo sarai in grado di

... sapere cos'è un insieme e quali sono le operazioni e le definizioni fondamentali sugli insiemi

... sapere cosa è il tipo SET in PASCAL e come si collega con quanto visto sugli insiemi

... sapere come si dichiara una variabile di tipo SET

... sapere come si costruisce concretamente un SET in PASCAL (inizializzazione di un SET)

... fare dei concreti esempi di insiemi e tradurli in statements di PASCAL

... sapere come si opera sugli insiemi in PASCAL per costruirne di nuovi, dati degli insiemi di partenza

... sapere come funzionano gli operatori relazionali nel tipo SET

... sapere come funziona l'operatore IN (test di appartenenza)

... usare concretamente i SET in PASCAL

IL TIPO SET

La definizione dei manuali è precisa, asettica, e icomprensibile al 90% dei comuni mortali.

"il tipo SET definisce l'insieme dei valori che è l'insieme potenza del tipo base".

Detta così non dice molto, a meno che le persone che leggono non siano un po' al corrente delle nozioni di insiemistica moderna. Sarà bene quindi parlare un po' di **cosa** è un insieme, prima di poter dire come lo si descrive e si usa in PASCAL.

Vi ricorderò anzitutto che l'insieme è un concetto base in matematica: con questo si intende che è uno dei cosiddetti "concetti primitivi", i quali non si appoggiano ad altri concetti che logicamente vengono prima di loro: un po' come il punto e lo

spazio in geometria. Data quindi una proprietà comune di un certo numero di elementi (che possono essere numeri, o uomini, o colori) questi elementi, presi nel loro complesso si diranno costituire un **insieme** (set). Occorre quindi, per poter parlare di un insieme, essere in presenza di una proprietà comune, magari di fronte a quella, estremamente generata, di "oggetto" o "ente". Potremo così parlare dell'insieme di tutti i gatti, e di tutti i gatti neri. E poichè un gatto nero è necessariamente un gatto, ecco che diremo che l'insieme dei gatti neri è **contenuto** nell'insieme dei gatti "O" anche che è un **sottoinsieme** (subset) dell'insieme dei gatti.

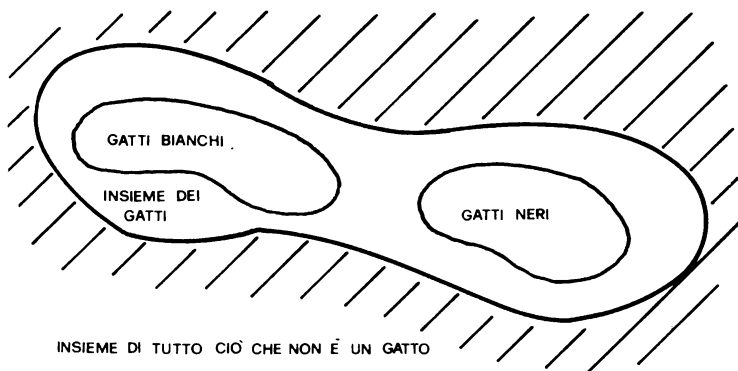
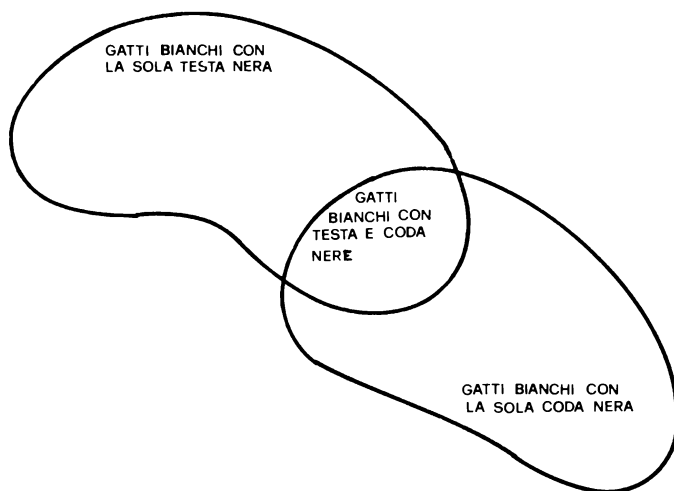
E potremo anche definire l'insieme dei gatti bianchi. Anche questo è un sottoinsieme dell'insieme dei gatti. A questo proposito possiamo vedere una cosa: i due sottoinsiemi (gatti bianchi e gatti neri) non hanno alcun elemento comune, cioè non esiste nessun gatto che possa essere contemporaneamente tutto bianco o tutto nero. Diremo allora che questi insiemi sono **disgiunti**, o che la loro **intersezione è vuota** o che il loro **prodotto è vuoto.**

Continuando l'esempio dei gatti, potremo costruire ora due altri insiemi:

I) gatti bianchi con la sola testa nera
II) gatti bianchi con la sola coda nera

E stavolta l'intersezione dei due insiemi non è detto che sia vuota: possono benissimo esistere dei gatti che hanno **sia** la testa **che** la coda nere. Questi gatti quindi appartengono ad entrambi gli insiemi: diremo quindi che appartengono all'insieme **intersezione** ed in questo caso l'intersezione (detta anche **prodotto**) dei due insiemi non sarà vuota, essendo costituita appunto da tutti quei gatti che si trovano nella condizione di essere bianchi in tutto fuorchè nella testa e nella coda.

L'insieme di tutti i gatti che sono bianchi, ma che hanno nere **o** la testa **o** la coda, **o** entrambe si chiama anche insieme **unione** o **somma** dei due insiemi di partenza.

La cosa può venir schematizzata molto agevolmente tramite i cosiddetti "diagrammi di VENN", che altro non sono che degli schizzi molto utili per far comprendere il gioco delle intersezioni per così dire "a colpo d'occhio". Qui di seguito riporto alcuni di questi diagrammi, proprio sull'esempio degli insiemi che siamo venuti considerando.

In PASCAL è possibile definire un tipo nuovo di dati strutturati (appunto il tipo SET), in cui si costruiscono proprio degli insiemi di elementi che per proprietà comune hanno la ovvia proprietà comune che ci può essere in un linguaggio di questo tipo: appunto il TYPE.

I "sets" in PASCAL sono quindi insiemi di elementi (**non** ordinati) che sono tutti dello stesso tipo. I tipi però non possono essere qualunque: possono — è vero — essere anche tipi definiti dall'utente, ma devono essere solo del genere **scalare** o **subrange**. Non sono quindi permessi "sets di arrays" o "sets di sets", e questa limitazione non è concettuale, ma è dovuta esclusivamente alle limitazioni della memoria e della velocità dei calcolatori attuali.

Torniamo alla definizione iniziale del tipo SET (parola riservata, o metacostante che dir si voglia: attenti a non usarla per scopi diversi da quelli canonici!) e supponiamo di definire un tipo (che chiameremo PRIMARI) e che consiste di tre "colori" GIALLO, ROSSO, BLU. La definizione di tipo sarà qualcosa come

TYPE PRIMARI = (GIALLO, ROSSO, BLU);

Con i tre elementi di questo tipo, potremo costruire una serie di insiemi: eccoli

GIALLO
ROSSO
BLU
GIALLO, ROSSO
GIALLO, BLU
ROSSO, BLU
GIALLO, ROSSO, BLU
ϕ

dove l'ultimo è l'insieme non costituito da alcun elemento, detto anche l'insieme

vuoto e che va sempre messo nel conto.

Siamo quindi in presenza di un **"insieme di insiemi"** che si possono costruire a partire dalle variabili di tipo PRIMARI. E' a questo insieme di insiemi che si dà il nome di **"insieme potenza"**.

E dopo questa digressione la definizione iniziale acquista un significato preciso: definendo il tipo SET del tipo PRIMARI intendiamo dire che le "variabili" di questo tipo potranno essere "elementi dell'insieme potenza", cioè **uno qualunque degli insiemi più sopra elencati.**

LA SINTASSI

Restiamo sempre col nostro esempio di tipo base "PRIMARI" definito precedentemente.

Vediamo ora il BNF che si deve usare per definire il tipo SET. Anzitutto ci sono due modi per definire il tipo SET. Nel caso sia stato precedentemente definito il tipo base con una dichiarazione di TYPE abbiamo questa dichiarazione di SET:

⟨dichiarazione di SET⟩ : : = TYPE ⟨identificatore⟩ = SET OF
⟨tipo di base⟩ ;

che, tanto per restare nel nostro esempio dei colori primari, diverrebbe (supponendo di chiamare PRIMCOL il tipo "set di primari")

TYPE PRIMCOL =SET OF PRIMARI;

Da questo punto in avanti il dire che una variabile strutturata del tipo SET è anche del tipo PRIMCOL vorrà dire che questa variabile può essere uno qualunque degli insiemi elencati nel paragrafo precedente. La cosa è perfettamente analoga a ciò che succede per una variabile scalare, solo che ora il "valore" che una variabile di tipo SET può assumere è in realtà **un insieme di valori**: uno qualunque degli insiemi che si possono costruire a partire dal tipo di base.

Possiamo però anche definire direttamente delle variabili del tipo strutturato SET direttamente, con una dichiarazione del tipo VAR. Così per esempio

VAR SET1, SET2: SET OF PRIMARI;

ed in questo caso SET1 e SET2 vengono definiti immediatamente come insiemi delle variabili di tipo PRIMARI.

Nel caso invece avessimo definito il tipo PRIMCOL attraverso una definizione di TYPE avremmo dovuto scrivere:

VAR SET1,SET2 :PRIMCOL;

Le due tecniche sono perfettamente equivalenti e PASCAL lascia all'utente la scelta di quella preferita, a seconda di esigenze di ordine e leggibilità e di preferenze personali.

Con questo possiamo considerare finito quanto c'è da sapere su come si dichiarano i SET. Rimane ora da vedere come si costruisce in realtà un SET e quali son le operazioni che su di esso si possono effettuare.

COME SI COSTRUISCE UN SET

Una volta che una variabile sia stata definita di tipo SET e sia stato definito anche il tipo base, resta da vedere come si costruisce un particolare SET.

La cosa è analoga a quella che succede per le variabili di tipo scalare o per gli arrays o per i records. Per le variabili di tipo scalare anzitutto dobbiamo definire il fatto che — diciamo — N è di tipo INTEGER. Così

VAR N: INTEGER;

e poi — prima di poter usare concretamente N — dovremo assegnarle un valore. Magari così

N : =34;

Con i SET succede esattamente la stessa cosa. Ecco intanto la sintassi dell'inizializzazione di un SET:

⟨inizializzazione di un SET⟩ : : =
  ⟨identificatore⟩ : = [⟨lista di elementi⟩] ;

⟨lista di elementi⟩ : : = ⟨elemento⟩ {, ⟨elemento⟩} | ⟨nulla⟩

⟨elemento⟩ : : = ⟨espressione⟩ | ⟨espressione⟩ . . ⟨espressione⟩

Così potremmo scrivere, sempre restando nel nostro esempio:

```
SET1 : = [ ] ;
SET2 : = [GIALLO, ROSSO] ;
```

e ciò vorrebbe dire che l'insieme SET1 non ha elementi (è il cosiddetto "insieme vuoto"), mentre l'insieme SET2 è costituito dagli elementi (**non** ordinati!) GIALLO e ROSSO del tipo PRIMARI. Nota l'uso delle parentesi quadre per definire gli elementi che appartengono ad un insieme.

Ma possiamo avere anche altri tipi di definizioni. Se SET3, SET4, SET5, SET6 sono insiemi del tipo base INTEGER ecco una serie di possibili inizializzazioni di questi insiemi:

```
SET3 : = [3, 7, 86, 5, 102, –7] ;
SET4 : = [5. .10, –7. .–3] ;
SET5 : = [J. .K+M] ;
SET6 : = [J–3. .J+3] ;
```

In questi casi SET 3 è costituito dai numeri elencati in parentesi quadre. Nota che **non** sono messi in alcun ordine. Questa però non è una pratica da seguire di solito, esclusivamente per ragioni di leggibilità e facilità di comprensione del programma. SET4 invece è costituito dai numeri 5, 6, 7, 8, 9, 10, –7, –6, –5, –4, –3. L'uso di una notazione del tipo di quella usata per il SUBRANGE evita una scrittura lunga e di difficile lettura. SET5 invece consiste di elementi che possono cambiare nel corso del programma: gli elementi che lo costituiscono partono da J ed arrivano a K+M **inclusi**. Cosa siano J e K+M dipende dal valore che essi posseggono al momento dell'inizializzazione del SET. Così SET6 consiste degli elementi

J–3, J–2, J–1, J, J+1, J+2, J+3

che possono variare — evidentemente — a seconda del valore di J. Infine SET6 può

essere costituito da elementi diversi in vari punti del programma, appunto a seconda del valore della variabile J.

Possiamo avere anche SET di caratteri. Supponiamo che SET7 sia definito come un insieme che ha CHAR per tipo base: ecco una possibile inizializzazione

SET := ['A'..'D', 'X'..'Z'] ;

SET7 sarà costituito dai caratteri

A, B, C, D, X, Y, Z

dove abbiamo abolito gli apici per facilità di lettura.

A questo punto dovresti avere abbastanza chiaro ciò che è necessario per avere una variabile (strutturata) di tipo SET pronta per l'uso:

I) **un tipo base**, che sarà il tipo di tutti gli elementi del SET (analogamente a ciò che succede per l'ARRAY) e diversamente da ciò che succede per il RECORD)

II) **una dichiarazione della variabile** (o di tipo e di variabile) che dica che il SET — diciamo MIOSET — è formato da elementi del detto tipo di base

III) **un'inizializzazione** (analogamente a ciò che succede per tutte le variabili di qualsivoglia tipo) che dica quali sono in concreto gli elementi che costituiscono l'insieme MIOSET.

A proposito del punto III) sarà bene farti notare che gli orientamenti attuali nella costruzione dei compilatori PASCAL sono nel senso che diventa **obbligatoria** l'inizializzazione **di tutte** le variabili che verranno usate nel programma. Non solo quindi occorrerà nel futuro dichiarare tutto ciò che si usa, ma pure dire quanto vale all'inizio. Ancora una volta è un irrigidimento ed una difficoltà apparenti. Avviene infatti molto spesso — soprattutto nel FORTRAN — che per avere dei compilatori "di manica larga" ci si trovi di fronte ad errori di programmazione estremamente difficili da scoprire, e quindi da correggere. L'orientamente di PASCAL (ed anche dei moderni linguaggi di programmazione ad alto livello) è invece per una "disciplina alla base", che eviti fin dove possibile dei guai futuri, per i quali è facile perdere varie settimane, alla caccia di errori.

LE OPERAZIONI SUGLI INSIEMI

Vediamo ora come si opera su questo tipo di variabili. E cominciamo subito a dire che si può operare con **due** tipi distinti di operazioni

I) con operazioni dirette sugli insiemi

II) con operazioni relazionali

Cominciamo con le operazioni sugli insiemi. Queste sono tre (fondamentali: poi queste possono essere combinate in vario modo fra di loro) e permettono di ottenere

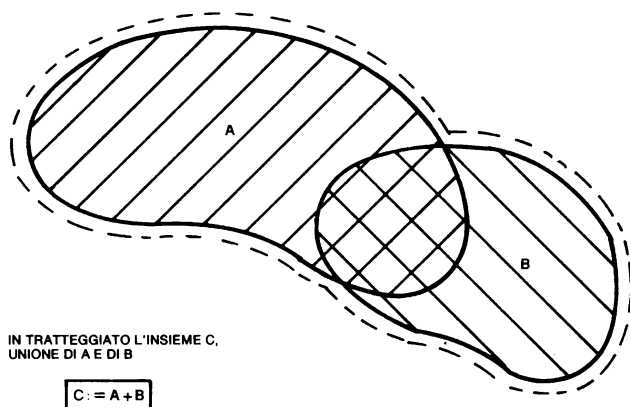
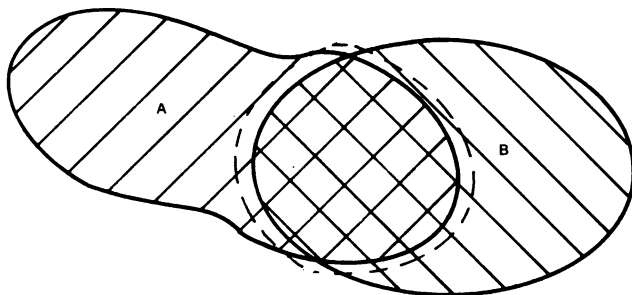
insiemi nuovi partendo da insiemi "operandi". E' importante rilevare (ma del resto penso che puoi già averlo pensato da solo) che gli insiemi operandi e l'insieme risultato delle varie operazioni devono avere tutti lo stesso tipo di base. Devono cioè essere tutti "insiemi omogenei".

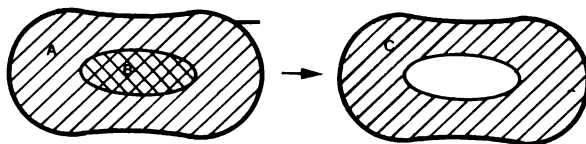
Ecco le operazioni fondamentali, coi relativi simboli degli operatori:

+ : unione di insiemi

* : intersezione di insiemi

— . differenza di insiemi

Penso sia bene indicare cosa queste operazioni vogliono dire in concreto con dei "diagrammi di Venn". Ecco quindi i diagrammi corrispondenti alle varie operazioni:

IN TRATTEGGIATO L'INSIEME C INTERSEZIONE DI A E DI B

C:=A*B

L'INSIEME C È L'INSIEME DIFFERENZA DI A E DI B (CHE DEVE ESSERE UN SOTTOINSIEME DI A)

C:=A−B

Facciamo un caso concreto: supponiamo che A sia l'insieme di tutti i numeri pari, e B sia l'insieme di tutti i numeri (sempre interi) compresi fra 0 e 100. In tal caso l'insieme unione di A e di B sarà costituito da tutti i numeri pari cui si aggiungeranno i numeri **dispari** fra 0 e 100.

Invece l'insieme intersezione di questi due insiemi sarà costituito dai soli numeri **pari** fra 0 e 100.

Supponiamo ora che A sia costituito da tutti i numeri interi fra 0 e 1000, e B sia costituito da tutti i numeri fra 0 e 1000 che risultano divisibili per 7. L'insieme B è evidentemente contenuto in A: potremo parlare allora di insieme differenza A–B e questo insieme sarà costituito da tutti i numeri fra 0 e 1000 **non** divisibili per 7.

Quindi tramite questi operatori noi siamo in grado di **costruire insiemi nuovi**, a partire da **insiemi "operandi"**.

Gli operatori relazionali producono invece come risultato una variabile booleana, a seconda del verificarsi o meno di certe condizioni. Vediamo di che si tratta, e supponiamo — al solito — che A e B siano due variabili di tipo SET:

A = B	TRUE	se gli insiemi A e B sono uguali (cioè contengono gli stessi elementi)
A<>B	TRUE	se gli insiemi A e B **non** sono identici. Nota che possono benissimo contenere **alcuni** elementi identici!
A<=B	TRUE	se A è **contenuto in B,** cioé se A è un sottoinsieme di B
A>=B	TRUE	se B è contenuto in A, cioè se B è un sottoinsieme di A
SCALARE **IN** A	TRUE	se SCALARE è una variabile di tipo scalare, dello stesso tipo base dell'insieme A, e risulta far parte degli elementi che compongono A. Per ragioni evidenti questo si chiama anche "test di appartenenza".

Vale la pena di chiarire alcuni punti:

I) non confondere le operazioni fra gli insiemi, che usano operatori aritmetici, con operazioni aritmetiche vere e proprie. L'uso degli stessi simboli è dovuto esclusivamente al fatto che le tastiere della macchina da scrivere sono limitate e che c'è un limite anche alla proliferazione dei simboli.

II) non confondere gli operatori relazionali che lavorano fra insiemi con quelli che lavorano fra numeri. Vale esattamente l'osservazione precedente.

III) un esempio sull'operazione **IN** (parola riservata!). Supponiamo che A sia l'insieme di tutti i **numeri pari fra 0 e 100.** In tal caso l'espressione

34 IN A

darà un valore TRUE, mentre

77 IN A   e   256 IN A

restituiranno entrambe dei valori FALSE. Perchè?

IV) Le operazioni sugli insiemi vanno sfruttate fin dove possibile. A parte il fatto del guadagno in chiarezza e concisione va sottolineato anche che queste operazioni sono particolarmente rapide per il calcolatore. Quindi familiarizzati con questo tipo di formalismo, perchè i programmi che scriverai guadagneranno in chiarezza concisione e velocità. Il che non è poco.

QUALCHE ESEMPIO

Supponiamo di dover dividere delle pratiche o altro a seconda delle iniziali di una certa parola (per esempio un cognome). E supponiamo anche di volerle dividere in quattro gruppi: dalla A alla K, dalla L alla O, dalla P alla S e dalla T alla Z. Ce la potremmo cavare con delle IF concatenate, e se supponiamo che la variabile su cui vogliamo effettuare la discriminazione si chiami INIZ, potremmo scrivere

```
IF INIZ>='A'  AND  INIZ <='K'  THEN .....
IF INIZ>='L'  AND  INIZ <='O'  THEN .....
IF INIZ>='P'  AND  INIZ <='S'  THEN .....
IF INIZ>='T'  AND  INIZ <='Z'  THEN .....
```

e la cosa funzionerebbe senz'altro. E' però molto più rapido per il calcolatore e più evidente per chi legge definire all'inizio degli insiemi del tipo CHAR in questo modo

```
VAR INIZ: CHAR; AK, LO, PS, TZ: SET OF 'A'..'Z';
```

e quindi avere dei test di questo tipo

```
IF INIZ IN AK THEN ....
IF INIZ IN LO THEN ....
IF INIZ IN PS THEN ....
IF INIZ IN TZ THEN ....
```

mentre gli insiemi AK, LO, PS e TZ vengono inzializzati con le

```
AK  :=  ['A'..'K']  ;  LO  :=  ['L'..'O']  ;
PS  :=  ['P'..'S']  ;  TZ  :=  ['T'..'Z']  ;
```

E se questo esempio ti sembra che non porti poi a grandissimi vantaggi prova a pensare a come faresti a separare le nostre ipotetiche pratiche a seconda che INIZ appartenga all'alfabeto italiano o sia uguale ad una delle lettere straniere J, K, W, X, Y. Col formalismo degli insiemi la cosa è immediata. Definiremo due SET di caratter ITAL e STRAN così

```
VAR INIZ:CHAR; ITAL, STRAN: SET OF 'A'..'Z';
```

e li inizializzeremo in questo modo

```
ITAL   :=  ['A'..'I', 'L'..'V', 'Z'] ;
STRAN  :=  ['A'..'Z'] - ITAL;
```

A questo punto i nostri confronti divengono semplicissimi: qualcosa come

```
IF INIZ IN ITAL THEN ....
IF INIZ IN STRAN THEN ....
```

Per confronto prova a scrivere gli stessi confronti con delle IF, in modo analogo a quanto fatto nel caso precedente. Ti accorgerai rapidamente della differenza di concisione e di leggibilità.

RIASSUNTO DI QUESTO CAPITOLO

⟨dichiarazione di SET⟩ : : = TYPE ⟨identificatore⟩
= SET OF ⟨tipo di base⟩ ,

⟨inizializzazione di un SET⟩ : : = ⟨identificatore⟩ : =
[⟨lista di elementi⟩] ;

⟨lista di elementi⟩ : : = ⟨elemento⟩ {, ⟨elemento⟩} | ⟨nulla⟩

⟨elemento⟩ : : = ⟨espressione⟩ | ⟨espressione⟩ ⟨espressione⟩

PROBLEMI ED ESEMPI

I) Fai almeno tre esempi di insiemi che tu conosci, definendo esattamente la loro unione e le loro intersezioni

II) Definisci con parole tue cosa intendi per insieme potenza e fai almeno tre esempi di insiemi e dei loro insiemi potenze.

III) Scrivi il BNF della definizione di SET. Commentala e spiegala

IV) Con riferimento al problema I) scrivi le dichiarazioni di SET degli insiemi che hai definito.

V) Prendi la lista delle sigle automobilistiche delle città italiane. Definisci un tipo nuovo che assuma questi valori. Costruisci poi un set di queste variabili per le sigle che appartengono alla regione Campania (od un'altra a tuo piacere). Inizializza il SET stesso con le sigle corrette, e spiega in dettaglio le istruzioni di PASCAL che sei venuto via via scrivendo.

VI) Con riferimento al problema V) costruisci il SET delle targhe che **non** appartengono alla regione Campania.

VII) Costruisci ed inizializza i SET delle targhe di Piemonte, Val d'Aosta, Lombardia e Liguria. Costruisci quindi il SET del Nord-Ovest, che comprende tutte queste regioni. Completa il tutto con tutte le parti dichiarative necessarie ed inserisci anche i commenti opportuni.

VII) Supponi che PIEMONTE, LOMBARDIA, LIGURIA siano gli insiemi delle targhe automobilistiche di queste regioni, e che NORDW sia l'insieme delle targhe automobilistiche del Nord-Ovest. Assegna il valore corretto alle seguenti funzioni booleane

```
PIEMONTE = LIGURIA
LOMBARDIA <> PIEMONTE
LOMBARIDA <> NORDW
LIGURIA <= NORDW
PIEMONTE >= LOMBARDIA
NORDW >= LIGURIA
CN IN PIEMONTE
GE IN LOMBARDIA
```

VIII) Si può fare un set di ARRAY? Perchè?

IX) Supponi di riprendere l'esempio delle targhe automobilistiche e di fare un SET in cui le targhe del Lazio sono ordinate basandosi sul numero degli abitanti delle città ed un secondo SET basato sull'ordine alfabetico. Chiamiamo questi SET SET1 e SET2. Che valore ha la funzione booleana SET1 = SET2? E perchè?

X) Scrivi un programma completo (dalla'intestazione, alle dichiarazioni, al blocco del programma stesso) che costruisca un SET dei numeri primi compresi fra 1 e 100.

CAPITOLO 13

IL TIPO FILE

SCOPO DEL CAPITOLO

Alla fine di questo capitolo sarai in grado di

. . . sapere cosa si intende per "file di dati"

. . . come si dichiara una file

. . . sapere come si dichiarano tipi strutturati il cui tipo base sia ancora un tipo strutturato

. . . sapere quali sono le operazioni fondamentali che si possono fare con PASCAL su una file

. . . sapere cos'è la variabile di buffer

. . . sapere come si opera sulla variabile di buffer di una file

. . . sapere perchè vengono introdotte le procedure standard READ e WRITE e come si usano

. . . sapere cosa sono le "files di testo" (text files)

. . . sapere le procedure standard sulle text files ed usarle

. . . sapere cosa sono le files standard INPUT ed OUTPUT

. . . operare concretamente sulle files tramite PASCAL.

IL TIPO FILE

Questo è un argomento che dovrebbe esserti più familiare dei precedenti se hai un po' di esperienza di programmazione: infatti questo genere di struttura è abbastanza frequente in tutti i calcolatori ed in tutti i linguaggi.

Vediamo quindi cosa si definisce come "file". Va detto anzitutto che "file" in inglese ha lo stesso significato di "fila" in italiano ("dati in fila", dunque), e che il termine di "file" risale ai tempi delle schede che venivano — appunto — "messe in fila", e lette — evidentemente — una dopo l'altra. Non si poteva accedere alla scheda 34 senza aver prima letto 33 schede.

Una file prende molto del concetto di "fila di schede", anche se — evidentemente — parecchie cose diverse ci sono.

Una file è un insieme **ordinato** di dati dello stesso tipo. Nota che i dati possono essere del tipo strutturato: si possono avere in altre parole delle files di arrays, o delle files di records. L'unica limitazione a questo riguardo abbiamo visto che vale per il tipo SET per il cui tipo base sono ammessi solo tipi scalari o subrange.

Dette così le cose si potrebbe pensare che non esiste poi una gran differenza fra arrays e files: entrambi sono insiemi ordinati di dati dello stesso tipo base, in fondo. Delle differenze però ci sono ed è bene averle presenti fin dall'inizio. Ecco quindi le differenze:

I) In ogni istante è possibile accedere ad un solo elemento della file. La file in altri termini è una struttura su cui si può operare "ad un elemento per volta".

II) è possibile accedere ad un nuovo elemento della file solo procedendo **sequenzialmente** nella file stessa;

Il punto II) è il più importante: mentre nell'array il tempo richiesto dal calcolatore per accedere ad un elemento è sempre lo stesso, indipendentemente dal fatto che questo elemento sia il primo o l'ultimo, ciò non succede nella file, nella quale la ricerca di un elemento procede sequenzialmente, e quindi occorre più tempo per accedere agli ultimi elementi della file, di quanto non ne occorra per accedere ai primi. E' per questa ragione che l'array è detto una struttura "ad accesso casuale" (Random Access Structure) mentre la file è detta una struttura "ad accesso sequenziale" (Sequential Access Structure).

Vediamo ora la sintassi della dichiarazione del tipo:

⟨dichiarazione di FILE⟩ : : =
    TYPE ⟨identificatore⟩ = FILE OF ⟨tipo della file⟩ ;

Così evidentemente sono dichiarazioni valide

```
TYPE FILAMIA = FILE OF SCALARS;
TYPE FILATUA = FILE OF REAL;
TYPE FILASUA = FILE OF PACKED ARRAY OF ['A'..'Z'] ;
oppure = FILE OF STRING;
```

l'ultima file essendo costituita da stringhe di caratteri alfabetici soltanto.

E' evidente che le files sono usate soprattutto quando c'è da interagire con memorie periferiche al calcolatore (nastri, dischi e simili) e che ci potranno essere limitazioni sulla lunghezza massima delle files, dipendenti dal particolare sistema implementato. Per quanto ci concerne, supporremo che nella memoria del calcolatore possano trovare posto "alcuni" elementi della file (di solito non può trovare posto **tutta** la file, che può essere anche molto grande) e che sia accessibile un elemento della file per volta: questo è in genere ciò che succede su tutti i sistemi implementati attualmente.

Abbiamo visto quindi come si dichiara una file, ed in cosa consiste. Vediamo ora come si individua un suo elemento.

In una file un elemento **non** può essere indirizzato (come accade per un array) o **individuato** (come succede per un record) o **controllato** per vedere se appartiene o no alla struttura (come succede col SET, con l'operatore IN). Invece con la stessa definizione di file, che supponiamo essere

TYPE TIZIO = FILE OF ARRAY [1. .10] OF INTEGER;

e che definisce una file i cui elementi sono costituiti da arrays di numeri interi, ognuno dei quali contiene 10 elementi, viene anche definita dal sistema stesso una cosiddetta **"variabile di buffer"** che nel nostro caso si chiamerebbe TIZIO↑ e che appartiene allo stesso tipo cui appartengono gli elementi della file, cioè al tipo base. Nel nostro caso quindi il sistema costruisce automaticamente la variabile TIZIO↑ che è un array di 10 numeri del tipo INTEGER.

A che serve la variabile di buffer? A mettere nella memoria del calcolatore l'elemento della file che di volta in volta si preleva ed a cui si fa riferimento. In altri termini: dalla file TIZIO si può prelevare solo un elemento per volta e solo questo elemento è disponibile nella memoria del calcolatore per poterci operare su. Questo elemento viene **sempre** chiamato TIZIO↑.

Possiamo fare un'analogia con certi juke-boxes nei quali il braccio destinato a prelevare il disco prescelto scorre a lato di una rastrelliera (la file, appunto) su cui sono conservati tutti i dischi: arrivato al disco prescelto il braccio lo preleva e lo mette sul piatto del giradischi (la memoria del calcolatore). Sul piatto del giradischi si trova dunque sempre o nessun disco o un solo disco per volta. A parte il fatto che se si vuole cambiare disco occorre prima mettere a posto il vecchio, mentre nel caso della file non è necessario "rimettere a posto" niente, visto che un elemento della file si **legge**, ma non si **preleva** (la file non viene modificata dall'operazione di lettura, mentre il contenuto della rastrelliera sì), l'analogia può continuare. Per cambiare disco bisogna che il braccio prelevatore scorra sulla parte rimanente della rastrelliera fino a che non ha trovato il nuovo disco. Se poi il selettore si accorge che il nuovo disco sta in una posizione precedente a quella attuale nella rastrelliera, il braccio selettore viene fatto riandare all'inizio della rastrelliera e la ricerca inizia da capo. Questo nei "vecchi" juke boxes, in cui il braccio selettore si muoveva in fase di prelievo solo in **una** direzione. Nei moderni il braccio si può anche muovere all'indietro — diciamo — di 5 posti e prelevare il disco che lì si trova. Questo però non succede nelle files, in cui il prelievo degli elementi può avvenire sempre e solo "in avanti".

OPERAZIONI SULLA FILE

La variabile di buffer può essere usata in una serie di operazioni standard che vedremo qui di seguito. Viene detta anche "finestra" ("window") proprio per questo suo carattere speciale, di assomigliare appunto ad un'immaginaria finestra che venga fatta scorrere lungo la file ed attraverso la quale si possa vedere solo un elemento della file per volta.

Le funzioni standard che vedremo non sono implementate in tutti i PASCAL del commercio. Si impone ancora una volta la osservazione che è necessario raggiungere al più presto un accordo sul PASCAL standard, in modo da evitare diffe-

renze che rendono la vita penosa nell'impellente necessità di farlo girare su un calcolatore diverso.

Vediamo ora quali e quante siano queste operazioni che possono venir effettuate sulla file. Le funzioni prendono anche il nome di "operatori di file" (file operators). La file immaginaria su cui penseremo di operare verrà chiamata ancora TIZIO.

EOF (TIZIO): Questa è una funzione che ritorna un valore booleano. Normalmente è FALSE. Ritorna un valore TRUE quando si tenta di porre in TIZIO↑ un elemento che non c'è, perchè la file è finita, quando cioè si vuole andare "oltre la fine della file". Funzione utilissima da usare magari in statements del tipo

IF EOF(TIZIO) THEN .....

per prendere decisioni diverse dalle solite quando ci si accorge di aver raggiunto la fine della File. La fine della file viene chiamata anche End Of File, o EOF per brevità.

GET (TIZIO): questa funzione pone in TIZIO↑ il prossimo elemento della file. Qui occorre un attimo di attenzione: se in TIZIO↑ si trova già l'ultimo elemento della file, una chiamata GET (TIZIO) (che corrisponde a voler ottenere un elemento inesistente, che si troverebbe **al di là** della fine della file) può dare **non** segnalazioni di errore, **ma porre in TIZIO↑ un valore non definito.** Anche questo dipende dal sistema, ma in genere è norma prudenziale assicurarsi di non essere alla fine della file. Per fare questo basta vedere cosa dice la funzione EOF, come abbiamo appena visto.

PUT (TIZIO): questa è un'operazione che è permessa solo se EOF è TRUE, cioè se ci troviamo alla fine della file. In tale caso TIZIO↑ viene **aggiunto** alla file. EOF resta ancora TRUE, ed il risultato è che abbiamo allungato la file di un elemento. E' bene notare esplicitamente che se si vogliono aggiungere elementi alla file è solo possibile allungare la file, cioè aggiungerli in coda. Non è permesso aggiungere elementi in mezzo alla file, cioè effettuare operazioni di inserimento. Proprio come dovrebbe succedere fra persone beneducate che siano in coda davanti ad una biglietteria.

RESET (TIZIO): se la file non è vuota (e non credere che sia una osservazione banale: può sempre capitare, magari per un non voluto errore) questa ritorna in TIZIO↑ il primo elemento della file. In altri termini la finestra si riposiziona all'inizio della file. Nel caso in cui la file fosse vuota (EOF sarebbe vera, in questo caso!) ciò che viene ritornato in TIZIO↑ può non essere definito. Dal che si evince essere ottima abitudine controllare cosa dice la funzione EOF prima di effettuare certe operazioni.

REWRITE (TIZIO): con questa operazione è possibile riscrivere l'intera file. EOF diviene TRUE, la fila TIZIO viene rimpiazzata con una file vuota ed il sistema è pronto per riscriverla daccapo. Questo è il procedimento che dovremmo usare se volessimo **inserire**

anche un solo elemento all'interno della file. Procedimento altamente sconsigliabile, ovviamente, e che si usa solo in casi estremi.

Nota che a causa della limitazione dei caratteri e delle tastiere puoi trovare la variabile di buffer anche indicate come TIZIO^, dove si usa l'accento circonflesso, qualora non ci sia il carattere "freccetta" sulla tastiera.

LE PROCEDURE STANDARD

Veramente non abbiamo ancora parlato di cosa siano le procedure, e quindi basterà accennare al fatto che sono dei programmi **che fanno parte di** un programma più vasto. Sono quindi dei "sottoprogrammi". Questo lo vedremo più in dettaglio più avanti. Per ora a noi interessa vedere alcune "procedure" che sono in tutti i sistemi PASCAL e che servono a rendere più facile il maneggio delle files e delle variabili ad esse legate.

Supponiamo quindi di voler leggere dalla file un suo elemento e di porlo nella variabile che chiameremo CAIO. In TIZIO↑ si trova un elemento della file. Se noi volessimo mettere in CAIO il prossimo elemento della file, dovremmo scrivere

GET (TIZIO); CAIO : =TIZIO↑ ;

ed è per ovviare a farraginosità di questo tipo che sono state introdotte le "procedure standard" che semplificano la scrittura notevolmente. Nell'esempio che stiamo seguendo basta scrivere

READ (TIZIO, CAIO);

senza alcun riferimento alla variabile di buffer, che è assolutamente inessenziale, quale "variabile di passaggio", per ottenere lo stesso scopo delle due istruzioni precedenti.

Una cosa analoga si può dire per la scrittura. Se volessimo scrivere qualcosa nella file, abbiamo visto che dovremmo scriverlo "in coda". Dovremmo quindi scrivere qualcosa come

TIZIO↑ : = CAIO; PUT (TIZIO) ;

col che avremmo allungato la fila di un elemento, aggiungendovi come ultimo elemento ciò che era contenuto in CAIO. Anche in questo caso possiamo semplificare la scrittura così:

WRITE (TIZIO, CAIO);

che è in tutto equivalente alle due istruzioni precedenti.

Le procedure standard servono quindi a rendere più facile ed intuitiva la scrittura di istruzioni che altrimenti diverrebbero eccessivamente lungagginose.

Nello stesso ambito si collocano delle istruzioni ulteriormente semplificate che servono a trattare le "files di testo" (text-files) di uso ancora più corrente, ed in cui un'ulteriore semplificazione è evidentemente la benvenuta.

Cosa sono le "text-files"? niente di più che files costituite da un unico tipo di elementi, e precisamente dal tipo CHAR. Ti ricordo che — a seconda delle implementazioni di PASCAL — il tipo CHAR può variare. Nelle implementazioni più complete il tipo CHAR è costituito da tutte le lettere dell'alfabeto inglese, da tutte le cifre e da una serie di simboli speciali (parentesi, segni di interpunzione, simboli algebrici. . .). Una file fatta di un tipo base CHAR quindi serve a descrivere proprio un "testo". Caratteri numeri e segni speciali compresi.

Ovvio che abbia un'importanza speciale, dal momento che l'entrata e l'uscita di comunicazioni del calcolatore tramite un cosiddetto "testo" consiste di variabili di tipo CHAR in varie combinazioni.

Questa è la ragione per cui in tutti i PASCAL è implementato un tipo standard detto TEXT, definito come

TYPE TEXT = FILE OF CHAR;

il quale tipo esiste in tutti i PASCAL esattamente come esistono i tipi REAL o BOOLEAN.

Una file di tipo TEXT si può quindi pensare come una lunga file di caratteri, lunga al limite quanto il testo stesso. In realtà è ben noto che quando noi scriviamo un testo non lo scriviamo "tutto di seguito", ma lo suddividiamo in pezzi più piccoli che vengono chiamati "righe" o "linee". Ciò succede anche in PASCAL, in cui sono previsti alcuni caratteri speciali per suddividere le files di testo in righe. Non importa quali siano questi caratteri, in quanto ciò dipende del particolare sistema implementato: normalmente sono il "ritorno carrello" (Carriage Return, o CR o) e il "salto di linea" ("Line Feed", o LF).

Se quindi volessimo scrivere un testo su una stampante, dovremmo prima creare una file di TEXT, con gli opportuni CR e/o LF inseriti nei posti giusti, e quindi cominciare a trasferire la file al di fuori del calcolatore, pronta per essere usata dalla stampante. Anche in questo caso vengono in aiuto alcune procedure standard, che evitano pesanti va e vieni di controlli ed istruzioni troppo dettagliate. Vediamo di cosa si tratta.

Anzitutto viene dato significato standard a due files particolari, che si chiamano INPUT ed OUTPUT. Tutte le operazioni di lettura da parte del calcolatore (se non è specificato il nome della file da cui stiamo leggendo) vengono fatte dalla file INPUT. E così le operazioni di scrittura vengono fatte nella file OUTPUT. Ciò evita di scrivere cose inutili, sottintendendo le cose ovvie fino a quando è possibile.

Così si scrive

EOF	al posto di EOF (INPUT)
EOLN	al posto di EOLN (INPUT)
READ(X)	al posto di READ (INPUT,X)
READLN	al posto di READLN (INPUT)
WRITE(X)	al posto di WRITE (OUTPUT,X)
WRITELN	al posto di WRITELN (OUTPUT)

dove il significato delle procedure è il seguente:

WRITELN (TIZIO): scrivi una riga nella file TIZIO (e vai a capo quando hai finito)

READLN (TIZIO): leggi una riga dalla file TIZIO.

EOLN (TIZIO): è una funzione booleana che diviene TRUE se si è raggiunta la fine della riga. Nella variabile TIZIO↑ viene posto un blank (spazio) e la posizione di TIZIO↑ corrisponde proprio al carattere separatore.

Penso che questo possa bastare come prima idea su ciò che si intende per maneggio di files in PASCAL. L'argomento lo riprenderemo più a fondo quando parleremo di Input e Output (I/O) in uno dei prossimi capitoli.

RIASSUNTO DI QUESTO CAPITOLO

⟨dichiarazione di FILE⟩ : : = TYPE ⟨identificatore⟩
= FILE OF ⟨tipo della file⟩ ;

PROBLEMI ED ESEMPI

I) Spiega in dettaglio cos'è una file e quali sono le differenze principali fra questo tipo di dato strutturato e gli altri che già conosci.

II) Scrivi e commenta il BNF di una file. Fai almeno tre esempi pratici di dichiarazioni di files e fai vedere come il BNF si applica a questi esempi.

III) Spiega cos'è la variabile di buffer.

IV) Dichiara una file di REAL. Spiega in cosa consiste la variabile di buffer e a cosa servono e quali risultati producono le funzioni EOF, GET, PUT, RESET, REWRITE.

V) Supponi di essere interessato alla statistica della statura degli italiani, e di voler quindi costruire una file di numeri REAL, ognuno dei quali rappresenti la statura di una persona. Scrivi un programma completo che esamini la file, conti il numero delle persone e faccia la media delle stature immagazzinate nella file. Ti serve in questo caso la funzione EOF? Come potresti fare nel caso tu volessi avere la possibilità di aggiungere nuove stature alla tua statistica?

VI) Spiega in dettaglio le operazioni di READ e WRITE

VII) Spiega in dettaglio le operazioni READLN e WRITELN.

VIII) Spiega in dettaglio cosa sono le text files e perchè ad esse viene attribuita una importanza particolare.

IX) Che cosa succede se tentiamo un'operazione di READ quando la file è finita?

CAPITOLO 14

IL TIPO POINTER

SCOPO DEL CAPITOLO

... sapere cosa si intende per variabile statica e dinamica

... sapere cosa si intende per tipo POINTER

... sapere come si collegano le variabili pointer alle variabili dinamiche

... sapere come si opera sulle variabili dinamiche con la procedura NEW

... sapere come si inizializza un pointer col valore NIL.

... essere in grado di costruire una catena di records, con l'uso dei pointers

.. sapere come si costruisce un archivio di dati tramite una catena collegata (linked chain) che usi i pointers

... essere in grado di effettuare operazioni di ricerca, inserzione, cancellazione e modifica di dati in un archivio, tramite l'uso dei pointers

IL TIPO POINTER (PUNTATORE)

Ci occuperemo ora di un particolare tipo di variabile, legata ai dati strutturati (tipicamente ai records, ma non necessariamente) con l'uso della quale si possono effettuare con estrema semplicità tutta una serie di operazioni tipiche della gestione di grandi masse di dati, quali ricerche in archivi, elenchi, statistiche, inserzione di dati nuovi, distruzione di dati vecchi, modifiche di dati in archivio, e così via. Queste operazioni si eseguono preferibilmente su memorie ad accesso diretto (dischi magnetici, ed ora anche memorie a bolle) e l'uso di questo tipo di dati diviene tanto più conveniente quanto più frequenti sono le operazioni di aggiornamento e di modifica delle banche di dati.

Hai visto che PASCAL offre al programmatore una varietà notevole di dati strutturati, ognuno dei quali ha pregi e limiti e quindi si adatta meglio di un altro a trattare un certo problema. A questo proposito è bene notare ancora una volta ciò su cui tutti i manuali insistono fino alla noia:

E' FONDAMENTALE che chi si accinge a scrivere un programma esamini per bene

il suo problema, e dopo un'attenta analisi, non "a naso", scelga il tipo di dati strutturati che più gli conviene

In linguaggi di vecchio tipo, come il FORTRAN, ciò non costituiva un problema, dal momento che di dati strutturati ce n'era uno solo (l'array), in cui al massimo potevanno variare le dimensioni. In PASCAL occorre invece porre molta attenzione al tipo di dati che si devono usare. Il premio per questa fatica supplementare consiste — al solito — in una maggiore chiarezza e concisione, oltrechè in una maggiore velocità di esecuzione.

Un esempio, proprio sul tipo POINTER. Questo è utilissimo quando si tratti di gestire "banche di dati", ed è tanto più conveniente quanto maggiori sono le operazioni di aggiornamento e di modifica della banca stessa. Qualora queste operazioni non avvenissero mai o molto di rado, allora il tipo POINTER è sconsigliabile, in quanto si traduce in una inutile complicazione del programma, ed in una maggiore lentezza di esecuzione. Questo è il caso — frequente nella ricerca tecnico-scientifica — in cui i risultati di esperimenti vengono conservati su dischi o nastri per essere poi analizzati con programmi elaborati apposta. In questo caso i dati della banca di dati non vengono affatto modificati, anzi: si cerca con tutti i mezzi di evitare che possano venire persi o distrutti. In questo caso il POINTER è sconsigliabile, ed essendo invece la velocità di ricerca di un dato particolare un parametro di particolare importanza, la scelta del tipo di struttura per dati di questo genere probabilmente potrebbe essere orientata verso l'array di records. In ogni caso tengo ancora una volta a sottolineare che questo "sesto senso" lo potrai imparare solo con la pratica e che nessun libro manuale te lo potrà mai insegnare.

Ritorniamo a considerare quanto si era detto a proposito delle files. Ti ricorderai che una file è una "sfilza" di dati tutti dello stesso tipo: quando viene definita una file CAIO PASCAL definisce automaticamente una variabile CAIO↑ nella quale può trovare posto uno ed un solo elemento della file. Questa variabile è concepita come una "finestra" che vede un solo elemento della file per volta e che può essere spostata su e giù lungo la file stessa, senza "salti". Se CAIO↑ è occupato dal II elemento della file e si vuole che venga occupato dal V bisogna prima farlo occupare dal III, poi dal IV, ed infine si può mettere in CAIO↑ il V elemento della file.

La finestra può insomma solo **essere fatta scorrere lungo** la file, senza salti. Inoltre non si possono aggiungere elementi alla file altro che in coda alla file stessa. L'alternativa è di riscrivere l'intera file.

E' evidente che la file si presenta come una struttura di dati piuttosto rigida, che non prevede se non sporadiche ed eccezionali modifiche, e l'uso della quale è da prevedersi praticamente solo in casi di accesso sequenziale o quasi-sequenziale. Tipico il caso in cui nella file sono contenuti dei dati statistici che non si prevede di modificare (per esempio una serie di cartelle cliniche di un ospedale), ma che possono venir arricchiti per aggiunte successive, e che in ogni caso verranno letti ed interrogati.

E' spontaneo chiedersi se qualcosa di analogo alla variabile di buffer (definita al momento stesso di definizione della file) non possa venir definito anche per strutture più complesse, riunendo così sia i vantaggi che offre la variabile di buffer, sia quelli offerti dall'uso di strutture più elastiche della file.

Per esempio, tornando al caso che avevamo visto nel capitolo che trattava dei records, e precisamente alla descrizione dei dati di un libro di una biblioteca tramite un record, c'è da chiedersi se anche in questo caso non potrebbe essere possibile definire un qualcosa di simile alla variabile di buffer di una file, in modo da permettere tutta una serie di operazioni che è da prevedere possano essere molto frequenti ed importanti nella gestione di un archivio di biblioteca. Operazioni quali la lista di tutti i libri che soddisfano a certe condizioni (stesso autore, oppure stesso rilegatore, o uguale casa editrice, oppure. . .); oppure aggiunta di nuovi acquisti all'interno dell'archivio, e non necessariamente solo in coda; eliminazione dell'informazione di un libro dall'archivio, una volta che sia sicuro che il libro è irrimediabilmente perso o distrutto; modifica dei dati di un libro, magari perchè ci si accorge che ci sono stati degli errori di battitura.

Possibilità di questo genere non sono offerte evidentemente da nessuno dei dati strutturati che abbiamo visto finora, ed è chiaro che sarebbero estremamente interessanti.

La possibilità di costruire dati di questo tipo c'è in PASCAL ed è associata ad un tipo particolare di variabile, che si chiama il tipo POINTER (puntatore). Prima di parlare del quale, dovremo però parlare delle caratteristiche delle variabili che possono apparire all'interno di un programma.

VARIABILI STATICHE E DINAMICHE

Le variabili statiche non sono altro che le variabili così come le hai conosciute finora. Hanno una serie di caratteristiche che abbiamo imparato a conoscere. Anzitutto devono venir dichiarate in un blocco per poter essere usate in seguito, quindi il compilatore, al momento di tradurre il programma scritto in PASCAL in istruzioni di macchina, assegna ad esse una locazione di memoria. Queste variabili possono venir richiamate, lette, modificate, usate come operandi, ogni volta che ad esse si fa riferimento tramite il loro identificatore. La loro esistenza è dovuta al fatto che ad esse è stata assegnata una locazione di memoria all'interno di un blocco, ed esse "esistono" (cioè hanno una locazione di memoria assegnata) fino a tanto che il programma opera nel blocco in cui sono state definite. Sono insomma "legate al blocco di definizione": questa è la ragione per cui prendono anche il nome di "variabili statiche".

PASCAL ha la possibilità di usare un altro tipo di variabili diversamente dagli altri linguaggi ad alto livello. Questo tipo di variabili **non** viene dichiarato, e ad esse **non** ci si può riferire con identificatori, **non** essendo ad esse assegnata una particolare locazione di memoria. E' nel corso del programma, quando se ne presenta la necessità che a queste variabili viene assegnata una locazione di memoria. Le variabili allora "prendono ad esistere", e continuano ad esistere fin che la necessità perdura. Sono insomma delle variabili che appaiono e scompaiono a seconda delle necessità del programma. In questo senso vengono anche dette "variabili dinamiche".

Quindi per una variabile dinamica non esistono dichiarazioni esplicite, nè identificatori cui poter fare riferimento. Esistono invece delle procedure **standard** con le quali si possono generare variabili di questo tipo, o meglio assegnare ad esse una

locazione di memoria. In queste procedure si genera anche una seconda variabile, nella quale è contenuto l'indirizzo di memoria che è stato assegnato alla variabile dinamica appena creata.

Questa seconda variabile per così dire "punta" alla prima. Viene infatti chiamata una variabile di tipo POINTER, cioè "puntatore", e contiene — ripeto — l'indirizzo di memoria cui si trova la variabile dinamica appena creata.

Alla variabile POINTER si può assegnare un "valore" che si chiama NIL (parola riservata in PASCAL, ovvia abbreviazione del latino NIHIL, che sta per "nulla"): in tal caso il POINTER non punterà a nessuna variabile dinamica di alcun genere.

Vediamo ora come si collegano concettualmente fra di loro variabili dinamiche e POINTERS.

VARIABILI DINAMICHE E PUNTATORI

Riprendiamo l'esempio dei records che descrivono i libri, e supponiamo di voler introdurre un POINTER in questo caso particolare. Supponiamo di chiamare P questo pointer: avremo bisogno allora di una dichiarazione di questo tipo

```
VAR P : ↑LIBRO;
```

e con questa dichiarazione avremo creato una nuova variabile (P), il cui tipo è specificato nella dichiarazione, ed in questo caso è il POINTER associato alla variabile di tipo LIBRO. La quale a sua volta hai visto essere un record piuttosto complesso, con possibilità di parecchie varianti. Da questo momento in poi P si dirà il pointer **legato** (bound) al tipo LIBRO.

Da questo momento in poi il calcolatore assegnerà una locazione di memoria di una variabile di tipo LIBRO, quando ciò verrà richiesto dal programma, e P indicherà la locazione di memoria che sarà stata assegnata.

La variabile LIBRO in esame sarà d'ora in avanti accessibile al programmatore solo tramite il pointer P, unico legame fra una variabile dinamica come LIBRO ed il mondo esterno.

La variabile LIBRO (ricordati che è un intero record!) a questo punto si presenta perfettamente analoga alla variabile di buffer nel caso delle files. E per accentuare la somiglianza fra i due concetti essa viene chiamata P↑.

Come si può creare concretamente una variabile dinamica? Occorre invocare una procedura standard, che si chiama NEW (altra parola riservata, che non può essere usata per altri scopi!). In altri termini: ogni volta che si invoca NEW (P) viene creata una variabile dinamica di tipo LIBRO (cioè viene ad essa assegnata una locazione di memoria) e l'indirizzo di memoria in cui si trova viene assegnato a P.

Riassumendo:

VAR P : ↑LIBRO: Con questa istruzione viene definita una variabile P di tipo

POINTER. Essa conterrà d'ora in avanti solo un indirizzo di una variabile di tipo LIBRO, cui verrà assegnata via via una locazione di memoria a seconda delle necessità del programma.

NEW (P); Con questa istruzione viene assegnata una locazione di memoria ad una variabile di tipo LIBRO. Questa variabile si chiamerà P↑ ed in P si troverà il suo indirizzo di memoria. Quindi P punterà alla variabile P↑ di tipo LIBRO.

COME SI POSSONO USARE I POINTERS

Coi pointers si possono costruire delle strutture di dati estremamente articolate. Vogliamo ora vedere come ciò sia possibile.

Anzitutto riprendiamo l'esempio del record di tipo LIBRO e supponiamo di aggiungere al record un nuovo campo, nel quale inseriremo poi una variabile di tipo POINTER. Non ci sarà bisogno di molte istruzioni. Occorrerà aggiungere alle dichiarazioni di tipo qualcosa come

```
TYPE LINK = ↑LIBRO;
```

che definirà un nuovo tipo (LINK appunto) come un pointer legato (bound) alla variabile di tipo LIBRO.

Occorrerà quindi introdurre un nuovo campo nel record, che contenga appunto la variabile di tipo POINTER (cioè LINK, nel nostro caso). Chiamato SUCC (per abbreviazione di **SUCC**essivo) questo campo, nella definizione del record occorrerà aggiungere un nuovo campo:

```
SUCC : LINK;
```

A questo punto abbiamo finito, e vediamo come possiamo costruire ora una banca di dati che contenga la descrizione dei libri della nostra biblioteca, e che abbia tutti quei vantaggi di facile accesso e di possibilità di rapidi aggiornamenti che avevamo detto essere possibili all'inizio.

Sarà anzitutto necessario costruire una "catena di records" (linked chain of records). Per fare questo supporremo che i libri siano N e definiremo tre variabili di tipo LINK: le chiameremo PRIMO, P, PC (per Puntatore Corrente). PRIMO verrà inizializzato facendolo puntare fuori dalla catena: ecco gli statements:

```
TYPE LINK = ↑LIBRO;
VAR PRIMO, P, PC : LINK;
PRIMO := NIL;
```

La successione delle operazioni da effettuare sono ora le seguenti:

I) occorre allocare un record come variabile dinamica

II) occorre mettere nel campo SUCC del record appena allocato un puntatore. Nel record del primo libro metteremo NIL, nel record del II libro metteremo un

puntatore che punti al record del primo libro, e così via.

III) occorre aggiornare il puntatore, rendendolo pronto per l'uso col prossimo record.

Nota che il campo SUCC di un record LIBRO si chiama P↑.SUCC. Ecco la codifica di quanto siamo venuti dicendo:

```
PRIMO := NIL;
FOR K :=1 TO N DO
   BEGIN
   NEW (P);       P↑.SUCC :=PRIMO;       PRIMO :=P
   END;
```

OPERAZIONI D'ARCHIVIO

Finora il tutto può sembrare alquanto accademico: in fondo non abbiamo fatto altro che costruire una catena di records in cui un campo è un puntatore, che dà l'indirizzo a cui trovare il record successivo. L'ultimo anello della catena ha il puntatore che non punta da nessuna parte, e cioè ha il valore NIL.

Vediamo ora invece l'uso pratico di una catena di records così congegnata e supponiamo di voler fare le tre fondamentali operazioni di un archivio: la ricerca di un dato, l'inserzione di un dato e la cancellazione di un dato.

E vediamo anzitutto il problema della selezione: supponiamo che C sia un numero e supponiamo di voler cercare un libro che ha il numero di catalogo uguale a C. Potremo seguire il seguente medoto:

"puntiamo al primo elemento della catena. Fino a che il numero del catalogo non è uguale a C, mettiamo nel puntatore il valore che punta all'elemento successivo della catena. Una variabile booleana (TROVATO) ci dirà se abbiamo trovato o no un libro col numero di catalogo uguale a C".

Ecco una possibile codifica:

```
PLCOR :=PRIMO;  TROVATO :=FALSE;
WHILE PLCOR <> NIL DO
   BEGIN
      WHILE PLCOR↑.CATALOGO <> C DOPLCOR :=PLCOR↑.SUCC.
      TROVATO :=TRUE
END;
```

Supponiamo ora di voler aggiungere un libro all'archivio della nostra immaginaria biblioteca, e supponiamo pure che per svariate ragioni ciò non è possibile farlo semplicemente aggiungendo il relativo record "in coda" all'archivio, ma che invece è proprio necessario aggiungerlo dopo il record attualmente in memoria, che ha come puntatore – ricordiamo – PCOR.

Anzitutto dovremo dare diritto di cittadinanza alla variabile dinamica che conterrà il record del nuovo libro:

```
NEW (NUOVO);
```

sarà lo statement che dovremo usare, con il quale verrà riservato dello spazio in memoria per il nuovo record. E — nel caso fossero previste delle varianti — lo spazio sarà abbastanza grande per poterle contenere tutte. Il pointer associato a questo record sarà chiamato — appunto — NUOVO.

L'inserzione del nuovo record nella catena di records collegati è ora facilissima: basta giocare sui puntatori. Al puntatore del nuovo record si darà il valore che aveva prima il puntatore del record individuato da PCOR, mentre il puntatore di quest'ultimo verrà fatto puntare proprio al nuovo record inserito. Ed ecco la codifica:

```
NUOVO↑.SUCC  := PCOR↑.SUCC;
PCOR↑.SUCC   := NUOVO;
```

ed a questo punto il nuovo record entra a far parte della catena collegata di records.

La cancellazione di un elemento della catena di records avviene in modo analogo. A questo proposito giova ricordare il significato della freccetta messa dopo una variabile POINTER: con PCOR↑ noi intendiamo il **record** cui PCOR punta. In questo record il puntatore si trova nel campo chiamato SUCC, quindi nel campo PCOR↑.SUCC. E la cosa si può ripetere. Questo secondo puntatore ad un record che si chiama

```
PCOR↑.SUCC↑
```

nel quale si trova un puntatore, nel campo SUCC, cioè nel campo

```
PCOR↑.SUCC↑.SUCC
```

E via di seguito.

Dovrebbe esserti chiaro adesso che per distruggere un dato dalla catena di records basta scrivere

```
PCOR↑.SUCC := PCOR↑.SUCC↑.SUCC
```

perchè in questo modo il pointer del record PCOR↑ punta non direttamente al record successivo, ma a quello **dopo** il successivo. Attraverso il gioco dei puntatori si ha quindi il "salto di un record".

Con ciò abbiamo completato l'elenco delle operazioni che si possono effettuare su archivi di dati. Resta un'ultima operazione di cui non abbiamo parlato ancora, e che è la **modifica** di un dato nell'archivio. Questa però è un'operazione che si può facilmente realizzare in due passi, distruggendo prima dall'archivio il dato che si vuole modificare, e poi inserendo al suo posto il dato nuovo, modificato. Quindi la distruzione seguita dall'inserzione è equivalente a ciò che chiamiamo "modifica".

Questo tipo di struttura di dati permette quindi di ottenere facilmente tutte le operazioni di archivio normalmente usate. Questi vantaggi si pagano naturalmente con una maggiore lentezza di accesso ai dati di una catena, per cui occorre valutare attentamente la convenienza di usare questo tipo di struttura.

PROBLEMI ED ESEMPI

I) Spiega in cosa consiste la differenza tra variabili statiche e variabili dinamiche.

II) Spiega cosa vuol dire assegnare ad una variabile di tipo POINTER il valore NIL.

III) Spiega cosa è una variabile di tipo POINTER ed a cosa serve in una situazione in cui siamo in presenza di variabili statiche e dinamiche.

IV) Spiega in quali condizioni un pointer si dice legato ad un tipo. Inventati almeno cinque esempi di tipi con relativi pointers e scrivi tutte le dichiarazioni necessarie.

V) Spiega a cosa serve e come funziona la funzione NEW. Applicala agli esempi che hai fatto nel punto IV

CAPITOLO 15

LE PROCEDURE E LE FUNZIONI

SCOPO DEL CAPITOLO

Alla fine di questo capitolo sarai in grado di

. . . sapere cosa sono procedure e funzioni e a che servono

. . . scrivere l'intestazione di procedure e funzioni

. . . sapere come si passano i parametri necessari a procedure e funzioni

. . . sapere cosa si intende per "scopo" di una variabile o di un parametro

. . . sapere cosa vuol dire passare dei parametri ad una procedura o funzione per nome e per valore

. . . scrivere correttamente delle procedure o delle funzioni in PASCAL

I SOTTOPROGRAMMI

Non sottolineerò mai abbastanza l'importanza dei sottoprogrammi. Non è tanto una facilità offerta dai linguaggi di programmazione, quanto una vera e propria necessità della mente umana, che si trova molto più a suo agio nel controllare problemi non troppo complessi e nel collegarli via via in schemi più completi ed organici. E quindi il primo passo da fare nell'analizzare un problema è proprio quello di individuare tutti i "blocchi" che possono essere affidati a sottoprogrammi, i quali poi possono venir richiamati quando è necessario per compiere il loro lavoro.

Di sottoprogrammi in PASCAL ne esistono di due tipi: le procedure (procedures) e le funzioni (functions). Le due sono esattamente la stessa cosa, salvo che per un particolare: le funzioni producono come risultato del loro lavoro solo **un** valore di tipo scalare. Non ci sarebbe alcuna necessità di distinguere procedure e funzioni se non fosse per il fatto che le funzioni sono estremamente comode ed usatissime nel calcolo tecnico e scientifico, e quindi è molto consigliabile avere per esse un formalismo a parte, snello e pratico da usare.

I sottoprogrammi quindi rispondono a due richieste fondamentali: la prima è di rendere più leggibile, corto e veloce nell'esecuzione il programma complessivo. La seconda è di permettere a chi risolve un problema tramite un programma di control-

lare meglio la situazione globale evitando di scrivere programmi lunghissimi che divengono rapidamente illeggibili e difficilmente seguibili, anche quando siano scritti in chiaro da una stampante. Come regola generale sarà bene che ti ricordi che un programma, o sottoprogramma che sia, non dovrebbe superare le 30-40 righe di codifica complessiva. Piuttosto meno che più. Dal conto sono evidentemente esclusi gli statements dichiarativi, che a volte possono essere parecchi, e parecchio lunghi (pensa di avere un record complesso da dichiarare e ti convincerai da solo di quanto ho detto).

Dunque, nell'analisi di un problema occorre in PASCAL seguire alcune strade maestre

I) analizzare per bene i dati che dobbiamo usare e scegliere in modo oculato il tipo di struttura ad essi più adatta

II) analizzare bene il problema in sè stesso e spezzarlo fino a dove è possibile in sottoproblemi da affrontare con procedure a parte.

Tutto ciò si può fare alla buona su un pezzo di carta (che è bene non perdere!) su cui si "buttano giù" le idee, così come vengono, in modo molto simile a ciò che fa uno scrittore quando comincia ad avere idee per un racconto o per un romanzo. Ti sconsiglio in questa fase "ordine e metodo". Invece estro e fantasia sono le cose più preziose: magari corredate da carta e matita per catturare un'idea geniale, che — se non la inchiodi subito — non ti ritornerà mai più in testa per quanti sforzi tu faccia.

Quando questa specie di canovaccio, misto di formule, idee, strutture di dati, organizzazione del problema, spezzamento del problema stesso in sottoproblemi, è abbastanza consistente (può magari essere necessario riscrivere un paio di volte tutti gli appunti in modo più comprensibile, dal momento che è facile che i tuoi fogli ad un certo momento divengano un inestricabile groviglio di note, cancellature, rimandi etc.), quando ciò succede, allora, e **non prima**, viene il momento di passare alla fase operativa. Quindi

III) scrivere in chiaro, in modo leggibile e comprensibile, tutto quanto si è prodotto. Il mio consiglio è che saresti molto avvantaggiato dallo scrivere direttamente in inglese, usando il più possibile il frasario di PASCAL. Un inglese un po' "addomesticato", insomma. Appunto: "strutturato".

A questo punto comincia il vero lavoro di traduzione in linguaggio di programmazione di ciò che hai finora pensato. I passi sono quindi:

IV) stendere un diagramma di struttura di ciò che hai scritto nel punto III

Non abbiamo ancora parlato dei diagrammi di struttura, che sono destinati a sostituire i vecchi "diagrammi di flusso" (flow-charts) permettendo a chi scrive di "pensare in modo grafico", ma ti accorgerai della loro potenza nei prossimi capitoli, e se diverrai abile e pratico nel loro uso potrai scorciare tutto ciò che ho detto finora in un solo passo: nella stesura dei diagrammi di struttura che progressivamente potrai cambiare e raffinare a seconda delle idee che ti verranno. Proprio come fare degli schizzi a carboncino di un quadro, che ancora esiste solo nella testa del pittore.

Sui diagrammi di struttura può darsi che tu debba lavorare parecchio, facendoli e disfacendoli, riscrivendo tutto da capo, perchè ti è venuta un'altra idea, e così via. Comunque ancora una volta il consiglio è di usare il più possibile il frasario di PASCAL, che — oltre a tutto — è un modo molto naturale di esprimersi.

Arrivato ad un diagramma di struttura soddisfacente, ti troverai di fronte ad una sorpresa: il programma, se hai l'accortezza di usare i "modi di dire" di PASCAL, è praticamente già scritto. Prendere il diagramma di struttura e tradurlo in PASCAL dovrebbe essere una cosa semplicissima. In realtà nel diagramma è già scritto il programma, con le sue parti di logica, con le iterazioni e con la struttura a blocchi già predisposta dal diagramma stesso.

E' a questo punto che si apprezza tutto quel complesso di tecniche che poi va sotto il nome di "programmazione strutturata", il quale complesso permette di concentrare gli sforzi nella risoluzione concettuale del problema in esame, e di evitare e limitare al massimo lo sforzo puramente manuale di tradurre un problema in termini di linguaggio comprensibile al compilatore.

Vediamo ora la sintassi delle dichiarazioni di procedure e funzioni. Ecco, al solito, il BNF

```
⟨definizione di procedura⟩  : : =  PROCEDURE ⟨identificatore⟩
        ⟨sezione parametri formali⟩
           {;⟨sezione parametri formali⟩}  );

⟨definizione di funzione⟩  : : =  FUNCTION ⟨identificatore⟩
        ⟨sezione parametri formali⟩
           {;⟨sezione parametri formali⟩}  ): ⟨tipo del risultato⟩

⟨sezione parametri formali⟩  : : =  ⟨nulla⟩ |
         ⟨identificatore⟩ {,⟨identificatore⟩} : ⟨identificatore di tipo⟩
       {;⟨identificatore⟩ {,⟨identificatore⟩} : ⟨identificatore di tipo⟩}
```

ed al solito ecco alcuni esempi dell'intersezione di procedure e funzioni:

```
PROCEDURE PRIMA;
PROCEDURE SECONDA (X, Y, Z : REAL; AA, BB, CC, DD: INTEGER);
PROCEDURE TERZA (NN,MM, KK: INTEGER; AA, BB: CHAR; DD: BOOLEAN);
PROCEDURE QUARTA (N :ARRAY [1. .20] OF BOOLEAN);

FUNCTION UNO: INTEGER;
FUNCTION DUE (A, B, C: REAL):REAL;
FUNCTION TRE (K : ARRAY [1. .40] OF CHAR; KK:INTEGER) :BOOLEAN;
```

Ed esempi di intestazioni penso li possa continuare a trovare da solo.

La cosa importante da far rilevare è che per le funzioni non solo occorre dichiarare ciò di cui le funzioni possono aver bisogno nel corso del loro lavoro, ma anche il tipo del risultato finale (sempre di tipo scalare: la funzione genera **un solo** valore come risultato).

LA DICHIARAZIONE DELLE PROCEDURE E FUNZIONI

In un programma ti ricordo che dopo l'intestazione è necessario dichiarare una serie di oggetti che nel programma stesso verranno usati, e che l'ordine di queste dichiarazioni è tassativo: occorre dichiarare

I) i **L**abels

II) le **C**ostanti

III) i **T**ipi

IV) le **V**ariabili

V) le **P**rocedure

VI) le **F**unzioni

Vale la pena di chiarire ancora cosa si intende per "dichiarazione di una procedura (o di una funzione)". Se per esempio abbiamo una procedura che si chiama BEPPE, **non** basta dire che BEPPE è una procedura, ma occorre anche scrivere in dettaglio cosa è la procedura BEPPE. Cioè, nella fase dichiarativa, occorre **scrivere per intero** il sottoprogramma BEPPE: questo è ciò che si intende per "dichiarare la procedura".

Questo può mettere in imbarazzo in un primo momento chi è abituato a programmare in FORTRAN, per il quale un sottoprogramma è un qualcosa di decisamente separato dal programma principale, che viene scritto in sede separata, programmato separatamente, compilato separatamente, col risultato che il programma pincipale è in genere abbastanza corto e fa riferimento ai vari sottoprogrammi. Un programma in FORTRAN si configura un po' come una costruzione di pezzi separati che vengono cuciti insieme da un programma principale (il cosiddetto "MAIN PROGRAM") che è in grado di richiamare i vari elementi della costruzione quando ne ha bisogno.

Niente di simile in PASCAL: le procedure fanno parte integrale del blocco cui appartengono, al punto di essere inserite nella parte dichiarativa del blocco stesso. L'analogo nel FORTRAN sarebbe un MAIN nel quale **prima** di poter essere chiamate dovrebbero essere scritte completamente tutte le subroutines di cui il programma avrà bisogno.

Una procedura (o una funzione) è in realtà un programma in sè completo, che sottostà alle stesse regole e condizioni degli altri programmi. Ha quindi anche lui bisogno di una parte dichiarativa, che è esattamente identica formalmente a quella che abbiamo già vista per quanto riguarda il programma principale. Così anche all'interno di una procedura o di una funzione occorre dichiarare labels, costanti, tipi, variabili procedure e funzioni. In altri termini tutto ciò che la procedura o funzione dovrà usare in seguito.

Nota bene che in una procedura o funzione si possono benissimo dichiarare altre procedure e funzioni, che da questa debbono venir usate. Ancora una volta il gioco del "nesting" è perfettamente possibile. E — naturalmente — all'interno di una procedura le dichiarazioni seguono le stesse regole già viste, anche per quanto riguarda le dichiarazioni di procedure all'interno di una altra procedura.

Sarà bene a questo punto che tu rilegga il capitolo III e IV sulle dichiarazioni e

sulla struttura a blocchi; penso che una rinfrescata ai concetti fondamentali sarà utilissima.

Dobbiamo spendere alcune parole ancora sul gioco delle dichiarazioni. Supponiamo quindi che TONINO sia una variabile definita nel programma principale (e quindi, evidentemente, **prima** di dichiarare le procedure). TONINO quindi ha uno scopo **globale**, e una procedura che faccia parte del blocco in cui TONINO è stata definita non avrà bisogno di definirlo anche lei. Potrà usare questa variabile liberamente, in quanto, essendo dichiarata in un blocco ha accesso a tutto ciò che nel blocco stesso viene dichiarato.

Supponiamo ora invece che anche nella procedura ci sia la dichiarazione di una variabile TONINO. E' lecito chiedersi cosa può succedere in questo caso, e se per caso non sorga qualche conflitto nelle dichiarazioni. In questo caso la regola di PASCAL è molto semplice: anzitutto dichiarazioni di questo genere (multiple, per così dire) sono perfettamente lecite, ed anzi sono spesso consigliate; in secondo luogo nella procedura conta come dichiarazione di variabile **quella più recente**. Nel nostro caso il programma capirebbe — insomma — che il TONINO dichiarato nella procedura è la variabile con cui la procedura intende lavorare ed a questa assegnerebbe una locazione di memoria **diversa** da quella assegnata al TONINO dichiarato nel programma principale.

La cosa può ingenerare qualche confusione le prime volte, fino a che non si è acquistata una sufficiente pratica ed allenamento mentale, tali da riuscire a pensare ad una procedura o funzione come a qualcosa di "completo in sè" che ha accesso a ciò che è definito nel suo blocco, ma che ha anche la possibilità di definire lei ciò di cui ha bisogno.

Regola fondamentale dunque: conta la dichiarazione più recente.

PARAMETRI PER VALORE E PARAMETRI PER NOME

Abbiamo visto che nell'intestazione di una procedura sono presenti ciò che abbiamo chiamato "sezione parametri formali". In questo paragrafo intendiamo vedere di che si tratta.

Capita spesso di avere delle procedure di tipo generale che — proprio perchè sono di tipo generale — è bene non abbiano riferimenti col blocco cui appartengono. Questo è un caso molto frequente: è in questa situazione che occorre fornire alle procedure le quantità di cui hanno bisogno per lavorare, ed in modo indipendente dal blocco in cui sono inserite, proprio perchè le stesse procedure, senza cambiamenti, possano venir usate in programmi diversi. Si dice allora che alle procedure (o funzioni che siano: ti ho già detto che non esiste alcuna differenza fra procedure e funzioni, da un punto di vista formale e concettuale) vengono forniti dei **parametri**, e si dice anche che a questi parametri si fa riferimento "per valore".

Supponiamo quindi di avere una procedura QUAD che serva a risolvere delle equazioni di II° grado. Occorrerà di volta in volta fornire alla procedura stessa i coeffi-

cienti A, B, C dell'equazione stessa, e questo lo si potrà fare con un'intestazione del tipo

PROCEDURE QUAD (A,B,C: REAL);

A questo punto la procedura — al suo interno — potrà lavorare su A,B,C ed effettuare i calcoli opportuni. I parametri sono quindi forniti come **ingresso** alla procedura. Abbiamo quindi

parametri in ingresso ⇒ riferimento "per valore"

esempio: A, B, C, : REAL

Nel caso in esame (QUAD, cioè) ciò evidentemente non basta: occorre che la procedura produca dei risultati e che questi siano utilizzabili. Nel nostro caso a noi servirebbero altri due numeri reali, che siano le radici dell'equazione stessa. Chiamiamoli X1 ed X2. Non solo: ma ci servirebbe anche una terza quantità, (diciamo F) che può essere del tipo BOOLEAN tale che se F è TRUE ne consegua che X1 ed X2 sono radici reali. Se invece F fosse FALSE X1 ed X2 sarebbero ancora di tipo REAL, ma sarebbero interpretabili come parte reale e parte immaginaria delle due radici complesse e coniugate.

Abbiamo dunque bisogno di "parametri in uscita". Questi parametri si dicono "riferiti per nome", e la loro definizione nella sezione parametri formali è leggermente diversa. Ecco come suonerebbe l'intestazione della procedura al completo

PROCEDURE QUAD (A,B,C :REAL; VAR X1,X2: REAL; VAR F: BOOLEAN);

Nota l'apparizione di VAR che definisce X1, X2, F come variabili interne alla procedura stessa, quindi

parametri in uscita ⇒ riferimento "per nome"

esempio: VAR X1, X2: REAL;

Qui si impone un discorso che fa riferimento al modo in cui il compilatore genera all'interno della macchina i riferimenti per valore e per nome. Senza entrare nei dettagli, che non sono peraltro importanti, ti posso dire che il riferimento per valore può essere usato **solo** per comunicare dati **all'ingresso** di una procedura e che in genere è preferibile usarlo quando i dati da comunicare sono relativamente pochi (nel caso di QUAD la cosa andrebbe benissimo, per esempio). Nel caso però in cui occorra comunicare alla procedura dei dati di una certa complessità quali dati strutturati, allora il compilatore spende parecchio tempo nel generare gli indirizzi che collegano i parametri vari (al di fuori della procedura) con le locazioni di memoria presenti all'interno della procedura. Ed altro tempo viene speso al momento dell'esecuzione per passare da un indirizzo all'altro.

Da questo punto di vista invece il riferimento tramite parametri variabili è più veloce, ed è senz'altro da preferire quando la procedura deve passare dei dati di una certa complessità, quali appunto i dati strutturati.

Inoltre il riferimento per nome (cioè con parametri variabili) per il meccanismo stesso della sostituzione permette di far non solo uscire, ma anche entrare dei dati in una procedura. E da questo punto di vista è un po' un'"ancora di salvezza" per coloro che siano — diciamo — poco pratici in tutta la faccenda.

Evidentemente il problema dell'uscita dei dati non si pone per una funzione, la quale è essa stessa l'uscita: produce infatti un solo valore che è il risultato da comunicare al programma che l'ha chiamata.

COME SI CHIAMANO PROCEDURE E FUNZIONI

Finora abbiamo visto come si scrivono le intestazioni delle procedure e delle funzioni. La loro struttura complessiva appare poi alla fine come un qualcosa di questo tipo:

```
PROCEDURE ADELE (................);
   (parte delle dichiarazioni)
BEGIN
   (parte delle istruzioni)
END;
```

E' invalso inoltre l'uso di mettere un commento **dopo** la END; della procedura, in modo da rendere più leggibile il programma complessivo: così

```
END;   (*ADELE*)
```

Non è obbligatorio, ma è molto utile.

La dichiarazione della procedura è quindi un qualcosa che comincia con PROCEDURE e finisce con END;

Vediamo adesso come possiamo invocare una procedura per farla lavorare.

Supponiamo quindi di avere una procedura che è chiamata così

```
PROCEDURE BETTINA;
   (parte dichiarativa)
BEGIN
   (parte delle istruzioni)
END;   (*BETTINA*)
```

quindi senza sezione parametri formali. Ciò vuol dire che o la procedura lavorerà sulle variabili o costanti definite nel blocco di cui fa parte, o che non avrà bisogno di alcun parametro. Tipiche — in questo caso — le procedure che devono scrivere qualcosa di fisso.

Per chiamare la procedura basterà pronunciarne il nome. Così

```
BETTINA;
```

ed il controllo del programma passerà alla procedura, tornando automaticamente al programma principale quando in BETTINA sarà stata raggiunta la END;. Per esempio potremmo avere nel programma principale qualcosa come

```
IF K=O THEN BETTINA;
```

il che significherebbe che BETTINA verrà chiamata nel caso in cui risulti K=O. Altrimenti il programma procederà con le istruzioni successive, e BETTINA non sarà chiamata.

Supponiamo ora di avere una procedura un po' più complessa, che chiameremo CARLETTO la cui intestazione si presenti in questo modo

PROCEDURE CARLETTO (A, B, C: REAL; N:INTEGER);

con parametri definiti "per valore". Ecco qualche possibile chiamata

```
CARLETTO (3.5, 6, 7.78E-3, 4);
CARLETTO (X+Y, 4.6, SQRT (Z), 42);
CARLETTO (SQR(GIGI), SIN(ALFA), COS(ROMEO), D DIV DD);
```

dove tutte le variabili di cui abbiamo fatto uso devono — evidentemente — essere state già definite nel programma chiamante. Come vedi si possono passare come parametri anche delle intere espressioni. Una possibile chiamata potrebbe essere dunque qualcosa di questo genere

```
IF KK > O THEN BETTINA
ELSE CARLETTO (XX, SIN(X+Y), GIGI, 5);
```

ed il significato dello statement penso sia chiaro a tutti.

Vediamo ora un caso più complesso, e supponiamo che ora la procedura abbia sia parametri definiti per valore che parametri definiti per nome (cioè parametri variabili)

PROCEDURE DORINA (A,B,C:REAL; VAR X:INTEGER);

Questa è quindi una procedura in cui verranno forniti tre valori di tipo REAL e restituirà un valore X di tipo INTEGER. Va da sè che questa sarebbe stata meglio definirla come funzione, se questo è tutto quello che fa. Se però oltre a restituire X facesse qualcosa d'altro (per esempio modificasse qualche variabile del blocco) questo **non** sarebbe più possibile farlo: una funzione infatti può **solo** restituire **un** valore, e di tipo scalare, per di più. Non può fare altro, nè restituire un valore di una variabile di tipo strutturato. Ecco una possibile chiamata:

```
WHILE Z>6 DORINA (SIN(X), X,SQR(X), Z);
```

In questo caso DORINA verrà chiamata finchè risulterà Z > 6. Nota che il fatto che X sia un parametro variabile nella definizione di DORINA ed al tempo stesso faccia parte degli argomenti con cui DORINA viene chiamata **non** porta a conflitti di competenze. Al momento dell'esecuzione, cioè della chiamata a DORINA, la X è quella del programma chiamante, ed a DORINA vengono passati tre valori reali, che al suo interno vengono messi agli indirizzi corrispondenti ad A, B, C. La X definita come parametro in DORINA non ha nulla a che fare con la X del programma chiamante. La X di DORINA sarà il risultato di certe elaborazioni che la procedura farà: e questo risultato verrà messo nella variabile Z del programma chiamante, il quale — finito che DORINA abbia il suo compito — avrà a disposizione un valore di Z (che ovviamente deve essere definita nel programma chiamante stesso) per le sue elaborazioni ulteriori.

Vediamo ora come si chiama una funzione: supporremo che questa sia definita come

FUNCTION ELSA (A:REAL; N:BOOLEAN; X:ARRAY [1..15] OF INTEGER):REAL

Nota che la funzione **non** può produrre un array come suo output, ma un array può benissimo essere uno dei suoi parametri!

Ecco quindi una chiamata:

```
FFF :=ELSA (3.5 ,TRUE, MATRICE);
```

3.5 è un valore indiscutibile di tipo REAL. TRUE è di tipo BOOLEAN, mentre MATRICE deve essere definita nel programma chiamante come un array di interi di dimensione 15, ed FFF deve essere definito come di tipo REAL. Vedi che la chiamata ad ELSA consiste in un semplice statement di assegnazione: dopo che ELSA avrà calcolato il suo valore finale, questo verrà assegnato alla variabile FFF del programma chiamante.

Nota importante: prima di uscire da ELSA occorre uno statement di assegnazione ad ELSA, che dica – in parole povere – quanto ELSA deve valere. Una FUNCTION si presenterà quindi come qualcosa di questo genere:

```
FUNCTION ELSA (. . . . . .);
   (parte dichiarativa)
BEGIN
   (istruzioni del programma)
   ELSA :=. . . un'espressione
END;
```

Col che possiamo ritenere finita questa parte istituzionale su procedure e funzioni.

RIASSUNTO DI QUESTO CAPITOLO

⟨definizione di procedura⟩ : : = PROCEDURE ⟨identificatore⟩
(⟨sezione di parametri formali⟩
{ ; ⟨sezione parametri formali⟩ }) ;

⟨definizione di funzione⟩ : : = FUNCTION ⟨identificatore⟩
(⟨sezione parametri formali⟩
{ ; ⟨sezione parametri formali⟩ }) : ⟨tipo del risultato⟩ ;

⟨sezione parametri formali⟩ : : = ⟨nulla⟩
⟨identificatore⟩ { , ⟨identificatore⟩ } : ⟨identificatore di tipo⟩
{⟨identificatore⟩ { , ⟨identificatore⟩ } : ⟨identificatore di tipo⟩

CAPITOLO 16

PROCEDURE RICORRENTI INPUT ED OUTPUT

SCOPO DEL CAPITOLO

Alla fine di questo capitolo sarai in grado di

. . . sapere in cosa consistono le procedure ricorrenti

. . . usare le procedure ricorrenti in alcuni casi concreti

. . . sapere come si introducono i dati in PASCAL

. . . sapere come si estraggono i dati da un programma in PASCAL

. . . sapere come si controlla l'uscita dei dati e la loro stampa in PASCAL

LE PROCEDURE RICORRENTI

Questo è uno dei più potenti strumenti di PASCAL; e, per quanto possa non sempre essere consigliabile usarlo, dato che può diventare un po' dispendioso in tempo di calcolatore, coi moderni calcolatori, data la loro velocità media questo può essere un ostacolo facilmente superabile. Il vantaggio è di avere nelle mani un attrezzo veramente poderoso che permette di effettuare calcoli e problemi difficilmente affrontabili con altri mezzi.

Stiamo andando verso una "programmazione avanzata", e qualora tu ti sentissi ancora incerto puoi benissimo saltare questa parte, leggendoti invece la parte che riguarda l'ingresso ed uscita dei dati, parte che — ovviamente — bisogna conoscere invece molto bene. Se — come spero — questo non è il tuo caso, allora procedi pure tenendo presente che questo sarà un argomento decisamente impegnativo, in quanto stiamo toccando uno dei più recenti strumenti dell'informatica moderna. Ed anche gli sforzi di chi scrive, volti a rendere facili cose che so benissimo che facili non sono, non è detto siano coronati da successo. Se troverai delle difficoltà penso che non ci sia altra strada che rileggere e ri-macinare questo capitolo fino a tanto che ti diverrà ovvio e naturale.

Premesso questo, vediamo in cosa consiste una "procedura ricorrente".

Si indica con questo nome una procedura **che chiama sè stessa.** Questo è lecito in

PASCAL, ma non è lecito di solito in altri linguaggi. Occorre perþ stare attenti a non abusare di questo attrezzo dato che — come ti ho detto — il suo uso comporta una certa spesa di tempo di calcolatore, per cui spesso un'iterazione basata sugli statements iterativi che sono offerti da PASCAL risulta più rapida ed efficiente. A questo proposito sarà bene dirti subito che gli esempi che faremo rientrano proprio nella categoria in cui sarebbe sconsigliabile ricorrere alla procedure ricorrenti: c'è però il fatto che esempi in cui queste divengono praticamente necessarie sono piuttosto complessi e richiedono commenti piuttosto lunghi per essere apprezzati.

Cominciamo quindi col solito esempio del "fattoriale di N". Si definisce in matematica il fattoriale di N (e si indica con N!) una quantità che vale

$$0! = 1$$
$$1! = 1$$
$$2! = 1 \cdot 2$$
$$3! = 1 \cdot 2 \cdot 3$$
$$\ldots\ldots$$
$$N! = ((N-1)!) \cdot N$$

Quindi vedi che N! è definito come il prodotto di 1 per 2 per 3 . . . per N. E vedi anche che i matematici hanno un modo estremamente sintetico e spiccio di definirlo. **Tutta** la definizione del fattoriale è condensabile nelle seguenti formule

$$0! = 1 \qquad\qquad N! = N \cdot (N-1)!$$

La definizione del fattoriale di N si basa quindi sull'uso del fattoriale di N−1. A questo punto abbiamo una definizione "a scaletta" per cui, definito **un solo** fattoriale (quello di 0), ne vengono di conseguenza definiti infiniti. Cioè tutti.

Possiamo tradurre tutto ciò in un programma? La risposta è — ovviamente — sì. Vediamo quindi una possibile funzione che calcoli il fattoriale, seguendo proprio la formula così sintetica (ed indubbiamente astratta e di difficile comprensione, a prima vista), ma anche così potentemente sintetica, cara ai matematici.

```
FUNCTION FACT (N:INTEGER) : INTEGER;
VAR R:INTEGER;
BEGIN
  IF N > 0 THEN R := N * (FACT (N-1))
           ELSE R := 1;
   FACT := R;
END; (*FACT*)
```

Vedi che se N=0 (sarà cura di chi chiama questa funzione assicurarsi che N non divenga mai negativo, dal momento che il fattoriale di N non è definito per N minore di 0) FACT vale 1. Nel caso invece in cui N sia maggiore di 0,FACT chiama sè stessa passando come valore da usare il numero N−1. Ed il processo continua finchè risulterà che nell'ultima chiamata FACT verrà chiamata con argomento 0. Tutte le chiamate allora ritorneranno su sè stesse e produrranno il risultato R corretto.

Facciamo un esempio semplice, e supponiamo di chiamare

FACT(2), che vale $1 \cdot 2$, cioè 2

I) viene chiamata FACT(2). Nella IF risulta $N > 0$

II) viene chiamata FACT(1). Nella nuova IF risulta N > 0

III) viene chiamata FACT(0). Nella IF risulta N = 0

IV) FACT(0) ritorna il valore 1

V) il controllo ritorna a FACT(1) che calcola N · FACT(N−1).
Nel nostro caso N = 1, FACT(N−1) = 1, quindi FACT(1) ritorna il valore 1

VI) il controllo ritorna a FACT(2) che moltiplica N (cioè 2) per FACT(N−1) (cioè 1). Il risultato è 2. Ed a questo punto FACT(2) esce col risultato corretto, avendo finito il suo compito.

Sorge ora una domanda: come fa una procedura (o funzione che sia) a chiamare sè stessa, con **parametri diversi?** Non è che per caso ciò faccia fare confusione al programma? La risposta — evidentemente — è che non succede nessuna confusione. Ogni volta che la procedura viene chiamata essa lavora sui parametri che le sono passati dal programma chiamante, ai quali viene assegnata la locazione di memoria che hanno al momento della chiamata, ed è **come se** venisse chiamata una procedura o funzione del tutto nuova. E' il compilatore che pensa a ciò, e per copiare una frase di Bowles, nel suo "Problem Solving with PASCAL", è **come se** ogni chiamata ad una procedura o funzione ri-inizializzasse la procedura stessa: come se le varie chia mate fossero contraddistinte — diciamo — da un **colore** diverso. E' il compilatore che pensa al fatto che i vari "colori" non possano essere confusi fra loro.

Vediamo un altro esempio, anche questo risolvibile con mezzi più semplici e meno concettualmente raffinati di una chiamata ricorrente, oltrechè più efficienti da un punto di vista di tempo di esecuzione. Si tratta della cosiddetta "serie di Fibonacci", che ha molte applicazioni in svariatissimi campi. La serie di Fibonacci consiste di una successione di numero ognuno dei quali è la somma **dei due precedenti.** Ecco quindi la successione dei "numeri di Fibonacci":

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Come potremmo scrivere in forma compatta una simile serie? Chiamando F_1, F_2, F_3, F_4 ... i numeri di Fibonacci, ecco la solita definizione matematica ricorrente:

$$F_1 = 0, \quad F_2 = 1, \quad F_N = F_{N-1} + F_{N-2}$$

Ed ecco il programma in PASCAL che stavolta non commenteremo, lasciandoti questo compito allo scopo di fare un po' di pratica.

```
FUNCTION FIB (N:INTEGER) : INTEGER;
VAR M:INTEGER;
BEGIN
  IF N>1 THEN M := FIB(N-1) + FIB(N-2)
      ELSE
         IF N=1 THEN M := 1
         ELSE M := 0;
   FIB := M;
END; (*FIB*)
```

Altri esempi e problemi li potrai trovare alla fine del capitolo.

INPUT ED OUTPUT

Molto di quanto abbiamo già detto, parlando delle files ora potremo usarlo senza doverci ripetere da capo. Fai quindi riferimento a quanto detto nel capitolo delle file se troverai qualcosa che ancora non ti è chiaro.

Supporremo d'ora in avanti che valgano le seguenti dichiarazioni

```
VAR   C1,  C2,  C3,    :CHAR;
      K1,  K2,  K3,    :INTERGER;
      R1,  R2,  R3,    :REAL;
      S1,  S2,  S3,    :PACKED ARRAY OF CHAR
```

Nota bene: in molti PASCAL di oggi il PACKED ARRAY OF CHAR (di solito di dimensioni da 1 ad 80) viene chiamato anche STRING, ed è un tipo standard, in aggiunta ai tipi standard che già conosci.

Supponiamo ora di voler introdurre una costante intera. Uno statement del tipo

READ (K1);

farà sì che il programma leggerà dalla tastiera dei caratteri numerici e — molto importante — **il primo carattere non numerico che verrà incontrato segnalerà la fine del messaggio.** Così se volessimo far leggere al calcolatore il numero 345 dovremmo scrivere proprio 345. Se scrivessimo 34 5 nella memoria entrerebbe solo 34: il blank (carattere non numerico) segnerebbe la fine della trasmissione. Se invece battessimo degli spazi **prima** di battere il primo carattere numerico, questi verrebbero semplicemente ignorati.

Queste regole sono applicabili anche agli altri tipi di variabili elencate, salve restando le regole sintattiche per le costanti numeriche viste nei primi capitoli.

E' perfettamente lecito leggere valori appartenenti a tipi diversi, anche se occorre stare attenti per evitare che spazi o segni + e — giochino brutti scherzi a chi sta un po' disattento.

Diverso è lo statement READLN (leggi una linea): in questo statement può venir letta una serie di dati e lo statement è considerato completato quando tutti i valori in esso previsti sono stati letti e la linea è completata con un segno di "a capo": è il cosiddetto "carriage return", o "ritorno del carrello".

Sempre restando al caso di variabili intere, per semplicità, supponiamo di avere lo statement

READLN (K1,K2,K3);

Il programma si fermerà a questo statement attendendo che noi introduciamo i dati da tastiera. Se noi introduciamo dati come

34 45 5 (CR)

dove (CR) indica il tasto di **C**arriage **R**eturn, verrà assegnato il valore 34 a K1, il valore 45 a K2, ed il valore 5 a K3. Ma anche se avessimo battuto sulla tastiera

34 45 (CR)   5 (CR)

avremmo ottenuto lo stesso risultato: in questo caso infatti dopo il primo (CR) la lista **non** sarebbe completata ed in questo caso il programma capirebbe il (CR) **non** come la fine della trasmissione dei dati, ma come un qualunque carattere non nu-

merico: quindi alla stregua di uno spazio. Solo **dopo** l'introduzione del III dato (il 5) il (CR) viene interpretato come "fine della trasmissione", ed il programma è libero di proseguire col rimanente delle sue istruzioni.

Sono particolarmente importanti le funzioni EOF (End Of File) e EOLN (**E**nd **O**f **L**i**N**e): queste sono delle funzioni booleane che fanno parte di ogni sistema PASCAL, e che tornano molto comode in programmi iterativi. Vediamone l'uso specifico.

La funzione EOF è inizialmente FALSE e tale resta fino a che alla fine di un READ o di una READLN non viene premuto un tasto particolare, al posto di (CR). Nella maggioranza dei sistemi in uso oggigiorno tale tasto è in realtà la combinazione di **due** tasti: la lettera C ed il tasto CTRL (che sta per **C**on**TR**o**L**), per questo viene anche chiamata "Control–C". In altri sistemi il tasto è uno solo e si chiama ETX, come abbreviazione di **E**nd of **T**e**X**t.

L'uso di questa possibilità può essere particolarmente apprezzato in programmi interattivi, in cui si vuole continuare ad eseguire un ciclo fino a che il programma non riceva l'ordine CTRL–C. Per esempio così:

```
REPEAT
   (vari statements, incluse delle READ o READLN)
UNTIL EOF;
```

In questo esempio il ciclo continua a venir eseguito fino a che non viene pigiata da tastiera la combinazione CTRL–C.

Bowles fa giustamente notare che occorre stare attenti a loops di questo tipo: se in essi ci sono più di una READ si potrebbero dare dei casi in cui il pigiare CTRL–C per errore provoca dei disastri. In tal caso è bene analizzare la situazione ed eventualmente includere statements del tipo

```
IF NOT EOF THEN ......
```

per evitare guai.

EOLN è un po' diverso, e può essere usato in modo simile. Inizialmente EOLN è FALSE, e tale resta fino a che una READ finisce con un (CR). La prossima READ rende automaticamente di nuovo FALSE la funzione EOLN. Quindi la EOLN dice se la linea è stata completata o no. L'uso della funzione EOLN non è molto frequente, però è bene sapere che c'è, perchè può capitare di risolvere con essa delle situazioni intricate in modo rapido ed elegante.

Questo per quanto riguarda dati introdotti da tastiera. Può capitare invece di dover introdurre nel calcolatore dei dati che sono memorizzati su disco o su nastro. In tal caso occorre sapere dei dettagli che non possiamo dare qui, dal momento che per leggere qualcosa occorre sapere da dove e cosa si intende leggere. In generale quindi è necessario sapere

I) in che posto è memorizzata la file che si intende leggere (se nastro o disco o altro), e se ci sono – per esempio – **due** o più dischi, anche il numero del disco stesso. Occorre insomma anzitutto individuare l'apparecchio su cui il files si trova.

II) noto questo occorre sapere come si chiama il file che ci interessa.

La combinazione delle **due** informazioni è ciò che si chiama il "nome della file". Occorre a questo punto uno statement READ del tipo

```
READ ("nome della file", . . . lista delle variabili . . .);
```

Per come si scriva il nome della file occorre vedere il manuale d'uso del particolare calcolatore e del particolare sistema che si ha a disposizione. E' così possibile leggere una file da disco e metterla in un array nel calcolatore. Per esempio con statements del tipo

```
READ (DISCOMIO.FILAMIA, XXX);
```

dove possiamo supporre che la nostra file, consistente di 178 elementi risieda sul disco di nome DISCOMIO ed in questo nella file di nome FILAMIA, e che inoltre XXX sia un array proprio di 178 elementi. Questo permette di usare la struttura delle files nel momento in cui è più utile (cioè negli apparecchi periferici) mentre lascia all'array il compito di tenere i dati nel calcolatore, dal momento che l'accesso ai dati in un array è estremamente rapido.

Non sempre i vari PASCAL permettono un giochetto di questo genere: può anche darsi che tu sia obbligato a scrivere qualcosa del genere

```
FOR K :=1 TO 178 DO
   READ (DISCOMIO.FILAMIA, XXX(K));
```

leggendo dalla file **un solo** elemento per volta ed andandolo a mettere nel posto giusto dell'array. Non ci sono regole generali, ancora.

Abbiamo supposto tacitamente che la file venga identificata dal nome dell'apparecchio periferico e dal nome della file in esso, separati da un punto: questa infatti è un'usanza abbastanza generale.

Regole analoghe a quelle per gli statements di lettura valgono anche per gli statements di scrittura, però ci sono parecchie differenze che vale la pena di esaminare una per una.

Anzitutto PASCAL ha una limitata anche se essenziale maniera di presentare i dati. Allo stato attuale dello sviluppo del sistema non ci sono sofisticati sistemi di controllo e di presentazione dei dati come in FORTRAN. Non sappiamo dire, per ora, se questa che è veramente una scelta — per così dire — filosofica è destinata a restare nel futuro o no. Resta il fatto che la presentazione dei dati raramente usa tutte le sofisticazioni dei FORMAT del FORTRAN, vero linguaggio di programmazione all'interno di un altro linguaggio, è che però ciò che offre ora PASCAL è più che sufficiente per gli scopi di estrazione di dati dal calcolatore e della loro presentazione.

Anzitutto vediamo come si possono scrivere messaggi in PASCAL. Ciò che si usa è il tipo PACKED ARRAY OF CHAR, o il tipo STRING, che è la stessa cosa, ma che è molto più pratico da usare e che ormai è divenuto di uso quasi generale. Supponiamo che S1 ed S2 siano due variabili dichiarate del tipo STRING, o PACKED ARRAY OF CHAR che sia.

Queste variabili possono essere inizializzate per esempio così:

```
S1 :='VIVA LA JUVENTUS';
S2 :='ABBASSO IL MILAN';
```

A questo punto uno statement del tipo

```
WRITE (S1);
```

provocherà la scritta su display o stampante, a seconda di quello che è stato deciso di scegliere come apparecchio periferico principale

```
VIVA LA JUVENTUS
```

mentre, se avessimo due statements di seguito, come

```
WRITE(S1);   WRITE(S2);
```

ci troveremmo la scritta quasi incomprensibile

```
VIVA LA JUVENTUSABBASSO IL MILAN
```

Nota che **non** c'è alcuno spazio fra le due stringhe, come del resto non c'era nella definizione delle stringhe stesse. Se vogliamo separare le due scritte dovremo prevedere di porre degli spazi opportuni. Magari definendo la stringa S1 in questo modo

```
S1 := 'VIVA LA JUVENTUS' ;
```

dove alla stringa di prima è stato aggiunto uno spazio "in coda". I due WRITE di prima provocherebbero ora la scritta

```
VIVA LA JUVENTUS ABBASSO IL MILAN
```

su cui spero molti di voi non potranno dissentire.

Quindi gli statements WRITE **non** provocano l'andata "a capo" dopo che sono stati eseguiti. Una successione di WRITE causa la scrittura di seguito sulla stessa riga.

Se volessimo invece cambiare riga, dovremmo scrivere

```
WRITELN(S1);     WRITELN(S2);
```

ed a questo punto ci troveremmo di fronte ad un'uscita di questo genere:

```
VIVA LA JUVENTUS
ABBASSO IL MILAN
```

Questo per quanto riguarda le stringhe di caratteri. Non è però necessario definire le stringhe di caratteri a parte. Potremmo anche scrivere direttamente

```
WRITELN('VIVA LA JUVENTUS'); WRITELN('ABBASSO IL MILAN');
```

ed avremmo lo stesso output. Inutile dire che la seconda strada è più pratica ed evidente, e quindi è di gran lunga preferita.

Quando vogliamo scrivere dei numeri normalmente non dobbiamo specificare nulla. Il programma sottintende che verranno usate in uscita tante colonne quante sono necessarie per scrivere il numero nella sua precisione massima. Ciò va bene per gli interi, meno per i numeri reali per i quali questo sistema di presentazione spesso delle limitazioni inaccettabili.

Sono quindi previste due possibilità di controllo della presentazione dei dati.

Per i numeri del tipo INTEGER si può scrivere dopo il simbolo del numero da

scrivere il carattere: seguito da un numero intero. Così, per esempio

WRITE (K1:7);

In questo caso il programma riserverà alla stampa o all'output del numero 7 colonne, a meno che il numero stesso non sia più lungo di 7 colonne, nel qual caso il numero verrà scritto così com'è, ignorando semplicemente il 7. Il numero dopo i: prende il nome di "campo". Si dice in altri termini che all'output del numero K1 viene riservato un campo di 7 colonne.

La stessa cosa vale per il tipo CHAR, in cui si può specificare il campo voluto: il carattere che viene stampato verrà appoggiato **sullà destra**, mentre le rimanenti colonne verranno riempite di spazi (blanks). Se non si specifica nulla ad una variabile del tipo CHAR verrà riservata una colonna. Nei manuali troverai che questo sistema di **sottintendere** qualcosa, qualora nulla sia stato specificato si chiama anche il "valore di default" (default value).

Per i numeri reali la faccenda si complica, in quanto non basta specificare il campo in cui si vuole che il numero reale venga inserito: occorre anche specificare quante cifre si desiderano dopo il punto decimale. Così una scrittura del tipo

WRITE (R1:10:5);

genererà un output del numero reale R1 in un campo di 10 colonne ed il numero reale verrà presentato con 5 cifre dopo il punto decimale.

Questo — per ora -- è tutto quanto c'è da sapere su come PASCAL presenta i dati. Come vedi non è molto, nè ci sono grandi raffinatezze. C'è l'essenziale, questo sì: ma è logicamente da prevedere che se PASCAL avrà fortuna e si diffonderà uno dei primi perfezionamenti del sistema sarà proprio nella costruzione di un più completo sistema di presentazione dei dati in uscita dal calcolatore. Per ora bisogna un po' accontentarsi: francamente questo è uno dei limiti di PASCAL ed uno dei grossi pregi del FORTRAN, che si è venuto maturando in un buon decennio di uso costante da parte della comunità scientifica.

CAPITOLO 17

I DIAGRAMMI DI STRUTTURA

SCOPO DEL CAPITOLO

Alla fine di questo capitolo sarai in grado di

. . . sapere cosa sono i diagrammi di struttura

. . . conoscere i simboli che si usano nei diagrammi di struttura

. . . conoscere le regole per costruire un diagramma di struttura

. . . usare i diagrammi di struttura in alcuni problemi concreti

. . . vedere come dai diagrammi di struttura si possa arrivare naturalmente alla stesura di un programma in PASCAL

ADDIO ALLE FLOW-CHARTS

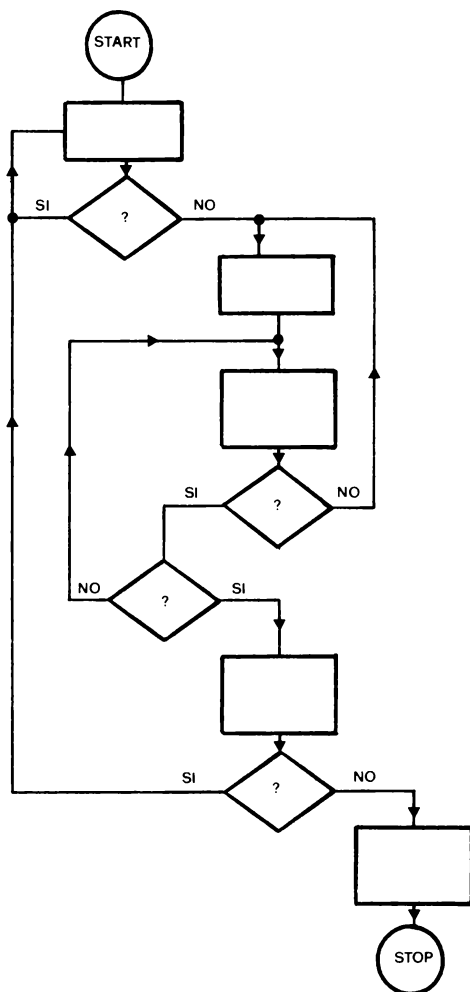
I diagrammi di flusso (flow-charts) sono stati l'argomento su cui hanno penato legioni di aspiranti programmatori, per buoni vent'anni. Ed è da riconoscere che con un linguaggio come il FORTRAN un diagramma di flusso è quanto di meglio si possa escogitare per visualizzare appunto il **flusso** delle istruzioni. In un programma concepito come una catena di istruzioni, con possibilità di saltare dall'una all'altra, spesso in modo difficilmente intuibile, un tipo di visualizzazione grafica che rendesse immediatamente visibile ciò che rapidamente diveniva un intricato groviglio, era per lo meno altamente auspicabile.

In linguaggi che invece concepiscono le istruzioni come blocchi separati (il famoso BEGIN . . . END, che vale per un'unico statement) congiunti fra loro da logiche immediatamente riconoscibili, le vecchie flow-charts non servono più. Peggio: rischiano di creare inutili confusioni e più lavoro di quanto ne risparmino.

Non starò qui a ricordarti cosa sono le flow-charts, perchè se le conosci è inutile, e se non le conosci è meglio che tu non ti confonda le idee ed impari direttamente i diagrammi di struttura. Basterà dire che si basano su due simboli grafici fondamentali:

I) il rettangolo in cui si scrivono delle operazioni da svolgere

II) il rombo (detto anche "diamante" (diamond)) per simboleggiare le decisioni logiche.

A queste si aggiungono via via vari simboli a scelta e per convenzione. Comunque, anche nella sua forma più semplice, è molto frequente che una flow-chart assuma aspetti simili a quelli in figura:

Puoi vedere da solo che la situazione non è semplice (in realtà ti posso assicurare che questo sarebbe un programma di **estrema** semplicità!) e che soprattutto una cosa confonde le idee di chi studi questo diagramma: **lo stesso punto può essere raggiunto da molti punti del programma stesso**. Ciò è esiziale ai fini della comprensione.

I diagrammi di flusso in realtà divengono rapidamente complicati oltre misura. E per **pratica** quasi ventennale di programmi di calcolo scientifico in cui ho praticamente fatto di tutto, dalla matematica avanzata a calcoli di tipica marca "gestionale" come si dice oggi, ti posso assicurare che è completamente falso ciò che viene regolar-

mente consigliato dai testi (analisi del problema, formalizzazione, stesura di un flow-chart, codifica vera e propria). Varrà per chi queste cose le insegna, ma non le ha mai fatte: per quanto riguarda me ed i miei colleghi (quelli che le cose le fanno, e non dicono o fanno credere di saperle fare, lasciando in buona sostanza che le facciano gli altri) quasi nessuno usa le flow-charts, a parte qualche caso di logica un po' intricata, o di stanchezza mentale. Sono estremamente dispendiose in tempo personale e non hanno quasi nessuna utilità. E' invece estremamente più utile essere capaci di scrivere un programma prima rozzo e poi sempre più raffinato, **e soprattutto** di centrare gli errori e le imperfezioni. Gli inglesi chiamano questa la fase di "debug": il "bug" è la cimice. Quindi "debug" è il "de-cimiciare": noi diremmo "spulciare", o "spidocchiare" se preferisci. La fase di "spidocchiamento" di un programma è quella in cui si vede veramente la padronanza del problema e della tecnica di chi lavora. Ed in questo caso ti assicuro che le flow-charts non servono a molto. E' abbastanza facile per noi passare a "spidocchiare" un programma (non solo, ma anche a perfezionarlo via via che si procede e che maturano nuove idee e magari nuove esigenze) per **150-200 volte** prima di essere soddisfatti del suo funzionamento. In tutto ciò mai una volta ho visto entrare il diagramma di flusso come fattore determinante su una scrivania.

Quindi non servono a nulla? No: ciò sarebbe esagerato. Servono quando uno muove i primi passi incerti ed è molto orgoglioso di riuscire a scrivere un programma che risolva le equazioni di II grado, naturalmente dopo accurata stesura di adeguata flow-chart. Ma appena si passa lo stato iniziatico il diagramma di flusso diventa praticamente inutilizzabile: il cervello da solo fa prima e meglio. E se deve essere impiegato in qualcosa di utile è meglio lo sia nella caccia all'errore. Arte difficilissima e che non si impara mai abbastanza. Esattamente l'analogo della "caccia al guasto" in un apparecchio complesso.

Un proverbio americano dice: "there's always one more bug". E' veramente la regola.

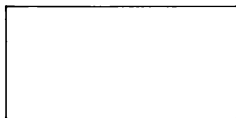
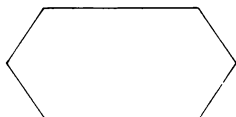
I DIAGRAMMI DI STRUTTURA

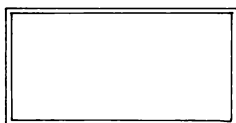
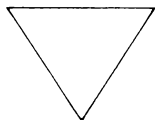
Al contrario dei diagrammi di flusso, questi permettono veramente di ragionare in modo grafico, scrivendo il programma in modo schematico e raffinandolo via via. Per cui nella creazione di un programma si può immediatamente cominciare a stendere un diagramma di struttura che verrà poi via via completato e reso più complesso, a mano a mano che chi scrive riflette sul problema trovando vie e soluzioni nuove e spingendo l'analisi del problema a dettagli maggiori.

Quindi – come consiglio personale – ti direi che di fronte ad un problema da affrontare con PASCAL dovresti **subito** cominciare a stendere un diagramma di struttura, per elementare e semplice che sia. E da questo momento in avanti non dovresti più abbandonare l'idea di **pensare per diagrammi**. Se possibile dovresti anche usare nel diagramma stesso la fraseologia di PASCAL, con i suoi WHILE, UNTIL, CASE, REPEAT etc. Questo ti aiuterebbe moltissimo nella fase finale, quando si tratta di passare dai diagrammi alla stesura del programma vera e propria.

Cominciamo a vedere i vari simboli del diagramma stesso: questi si chiamano "scatole" e sono di forma varia. Ad ogni forma corrisponde un'azione diversa da parte

del programma. All'interno delle scatole viene scritta l'azione che quella scatola è intesa compiere. Naturalmente i primi diagrammi che scriverai saranno molto elementari: più che altro un semplice pro-memoria di quello che una scatola è intesa fare. Quindi le azioni descritte nelle varie scatole saranno descritte in maniera molto sommaria e sintetica. Quindi saranno azioni **molto complesse**. Tipo "risolvi la tale e tal'altra equazione integro-differenziale". Man mano che procederai nell'analisi del problema le scatole verranno sempre più dettagliate, e descriveranno azioni sempre più **semplici**. Quando sarai soddisfatto del dettaglio raggiunto potrai passare a stendere il tuo programma direattamente in PASCAL.

Vediamo ora i simboli delle varie scatole.

IL RETTANGOLO: nel rettangolo si possono mettere tutte le operazioni di I/O e tutti gli statements di assegnazione. Quindi tutte le espressioni aritmetiche e non.

L'ESAGONO: in questa scatola possono essere messe solo le azioni che implicano una decisione logica. Quindi le IF dei vari tipi e lo statement CASE. Niente altro.

L'OVALE: in questa scatola possono essere messe solo le azioni iterative (quindi FOR, WHILE, REPEAT)

IL DOPPIO RETTANGOLO: in questa scatola vengono racchiuse procedure e funzioni, la cui descrizione dettagliata viene poi fatta a parte.

IL TRIANGOLO IN GIU': questo è un rimando ad un'altra zona del diagramma di struttura. In genere ad un'altra pagina.

IL TRANGOLO IN SU: questo è il complementare del precedente, e "riceve" il rimando da una pagina precedente.

IL CIRCOLETTO: serve a rendere possibile il collegamento di più scatole in una sola unità logica. Lo vedremo più avanti all'opera.

Continuiamo con le regole. Le scatole possono essere disposte **solo per successive file orizzontali**. Ognuna di queste righe prende il nome di "livello".

Le scatole che appaiono ad un livello sono delle azioni del programma che devono venir svolte in un ordine preciso. Tale ordine è stabilito nel senso che va **da sinistra a destra**.

Le scatole dei vari livelli sono collegate fra loro da linee che descrivono il passaggio di controllo del programma da una scatola all'altra. Sono **proibite** le linee che collegano scatole allo stesso livello. Si possono avere linee che collegano scatole solo fra **due livelli vicini**, ed ogni scatola di un livello inferiore può e deve ricevere **solo una linea** da una scatola al livello immediatamente superiore. Invece una scatola di un livello superiore può mandare **più linee** alle scatole del livello immediatamente inferiore.

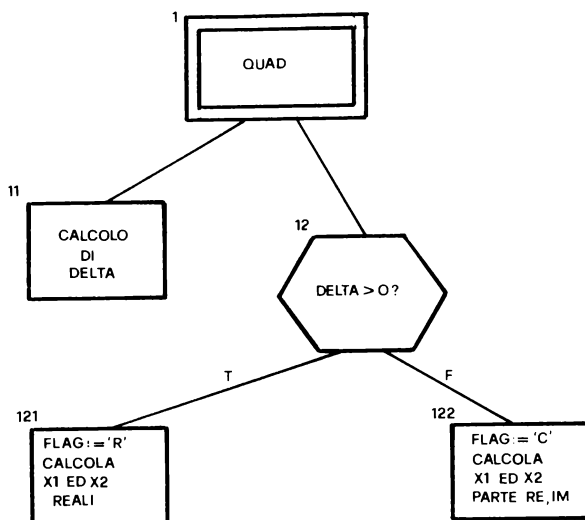
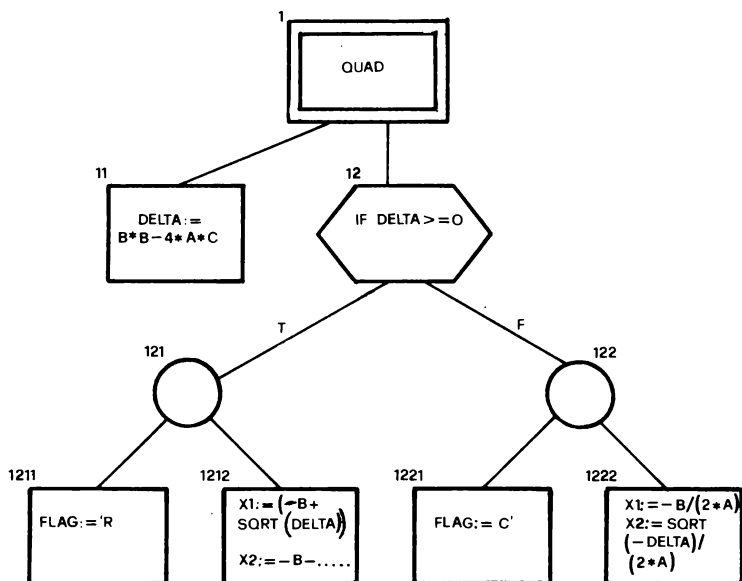
Il circoletto serve a rendere possibile la spiegazione in dettaglio di azioni complesse: **una sola linea** può raggiungere il circoletto, dal quale però possono dipartirsi quante linee si vogliono: queste vanno ad altrettante scatole. Quindi il circoletto permette di costruire un **livello supplementare** per descrivere più in dettaglio il programma. Da questo punto di vista è esattamente l'equivalente grafico del BEGIN ... END di PASCAL che per PASCAL è uno statement solo, mentre in realtà può essere un intero programma.

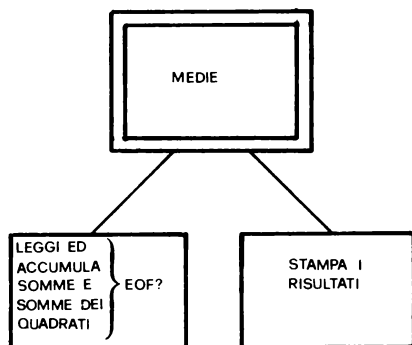
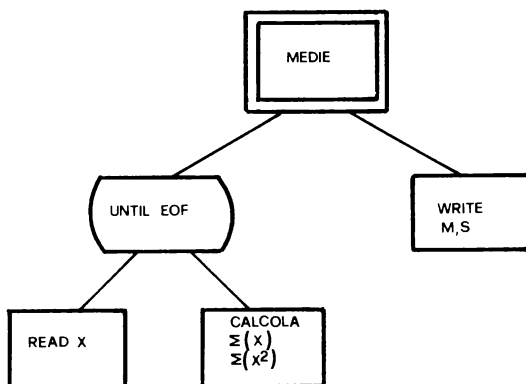
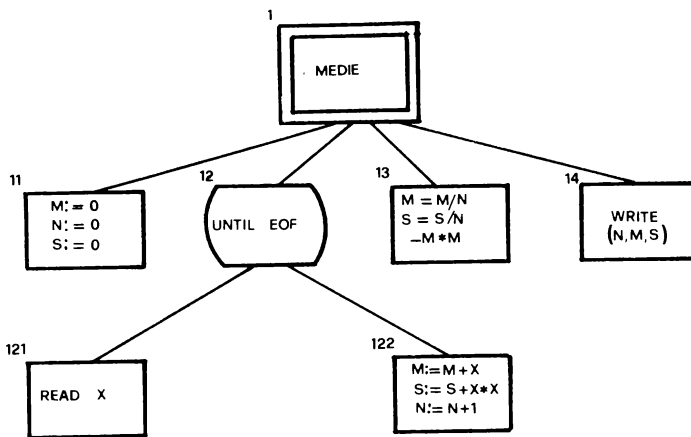
Da una scatola il controllo del programma passa alla prima (a sinistra) scatola del livello successivo. Quando il compito di questa scatola è **finito** il controllo ritorna alla scatola di partenza e da quella alla seconda scatola del livello successivo e via via con le altre. Per considerare **finito** il compito di una scatola occorre che tutte le azioni (anche di eventuali livelli successivi che a lei fanno capo) siano concluse.

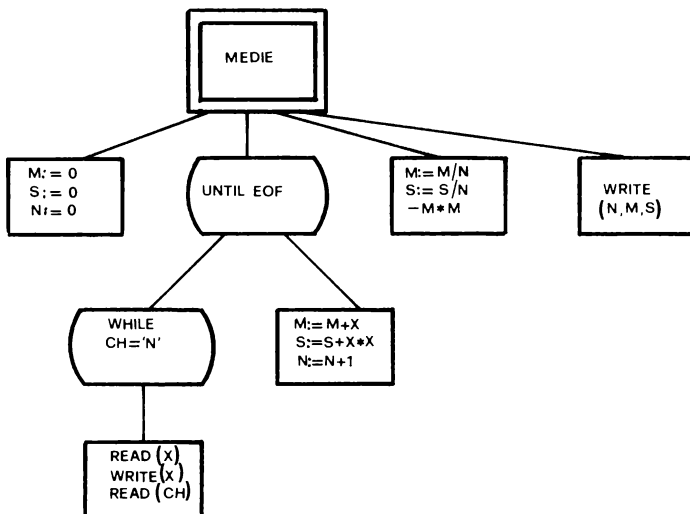
I livelli vengono numerati progressivamente con un sistema di aggiunta di cifre, simile al sistema decimale in uso nelle biblioteche.

La prima scatola si chiama livello 1. Quelle ad essa immediatamente collegate prendono i numeri 12, 13, 14 .... Se ci sono scatole collegate alla scatola 13, i corrispondenti livelli si chiamano 131, 132, 133 .... E così via.

Accanto alle varie linee di collegamento possono essere annotate le condizioni in cui queste linee vengono seguite: così, se si tratta di confronti, accanto alle varie linee possono apparire le diciture T e F per TRUE e FALSE. O cose più complesse se si ha a che fare con uno statement CASE.

Penso che a questo punto occorra passare ad un esempio concreto, e così nella figura puoi trovare un primo schizzo di ciò che potrebbe essere un diagramma di struttura per la solita procedura QUAD per la risoluzione delle solite equazioni di II grado. Da qui puoi vedere il gioco delle linee e dei livelli. Ed anche il fatto che le scatole sono lasciate volutamente un po' vaghe, in modo da indicare solamente le operazioni che esse sono intese effettuare.

Nella figuıa successiva puoi invece vedere lo stesso diagramma di struttura spinto nel dettaglio. E puoi anche vedere che a questo punto il passaggio alla codifica in PASCAL è estremamente facile ed immediato.

Va subito detto che questo è un esempio puramente accademico: l'esempio infatti è estremamente semplice proprio per capire come funziona il meccanismo. In realtà nessuno stenderebbe un diagramma di struttura per problemi così elementari.

Supponiamo ora di voler affrontare un problema più complesso: vogliamo essere capaci di introdurre da tastiera una serie di numeri dei quali il programma dovrebbe calcolarci la media e lo scarto quadratico medio. Però non sappiamo quanti numeri dovremo introdurre da tastiera. Quindi pensiamo di far ricorso alla possibilità di fare un test sulla funzione EOF. E cerchiamo anche di ragionare in modo grafico fin dall'inizio, commentando il meno possibile.

Schema 1

Schema 2

Lascio a te lo scoprire i successivi passi che sono stati usati nella stesura di questi diagrammi, fino al diagramma finale, che — come vedi — è ormai praticamente la codifica in PASCAL. Una sola breve nota a proposito della scatola 13: vi appare una formula che non tutti conoscono per il calcolo del cosiddetto "scarto quadratico medio". Esso si può ottenere oltre che facendo la differenza fra il valore della variabile X e la sua media M (il cosiddetto "scarto"), quindi elevando al quadrato e facendo la media delle quantità così ottenute, anche facendo la media dei quadrati di X e sottraendo alla fine il quadrato della media di X.

Supponiamo ora di volerci premunire contro possibili errori di battitura. La filosofia che seguiremo sarà quella di far scrivere alla macchina il valore di X appena battuto e di chiedere conferma: un qualunque carattere esclusa la N vorrà dire "va bene", una N vorrà dire che il dato era sbagliato e che occorre ribatterlo. Ecco il diagramma:

CONCLUSIONE

Con questo penso che tu abbia gli elementi in mano per scrivere un programma in PASCAL, partendo dalla formulazione del problema "a parole" ed arrivando alla codifica finale.

Ci sarebbero ancora alcune cose da dire a proposito di estensioni di PASCAL che sono state proposte e che probabilmente verranno accolte nel futuro, ma penso che questo vada al di là degli scopi di questo lavoro, che si è proposto solamente di riempire una lacuna attuale in Italia: dare ciò alle persone interessate uno strumento col quale esse potessero veramente usare nella pratica questo linguaggio di programmazione.

Un altro avviso, già peraltro ripetuto parecchie volte in questo lavoro: questo libro serve a ben poco, anche se seguito con la massima diligenza se non è accompagnato dall'uso pratico di un calcolatore. Ed anche un calcolatore serve a poco se serve solamente a fare qualche esercizio fine a sé stesso, o al massimo a programmare dei "giochi televisivi". E' la risoluzione del problema concreto quella che permette di lavorare sul serio e di imparare la tecnica.

Il calcolatore serve invece moltissimo per aiutare in tutti quei compiti di complessa segreteria che vanno sotto il nome di "calcolo gestionale", e serve soprattutto ad espandere le possibilità e le capacità di chi lavora in campo tecnico e scientifico.

Se questa riflessione spingesse molta genete (e soprattutto molti giovani) verso una riscoperta ed una riconquista di quella cultura scientifica che in Italia ha tradizioni altissime, pari solo all'attuale abbandono e decadenza anche e soprattutto nell'insegnamento scolastico (per non parlare del silenzio dei Grandi Mezzi di Comunicazione), ebbene, in questo caso sarei particolarmente orgoglioso di aver portato questo tipo di contributo.

Quindi se hai hobbies di tipo scientifico (sei astrofilo? ti interessi di radiofonia? elettronica? fotografia? biologia?) non avere paura delle formule: un calcolatore **non è** la Macchina Pensante o il Cervello Elettronico: queste frasi da giovanili ambizioni letterarie per fortuna frustrate lasciale pure a giornalisti e presentatori TV. Tu usa il calcolatore (se puoi affrontarne la spesa) per quello che è: un eccezionale attrezzo per permetterti di effettuare calcoli altrimenti al di fuori della portata di ogni essere umano, ed un sorvegliante continuo e diligente della tua strumentazione. In grado come tale di controllarla 24 ore su 24 e di prendere i dati per te e di elaborarli per te. Il prezzo da pagare è naturalmente che ogni attrezzo occorre imparare ad usarlo. Ma queste cose non sono poi più difficili del nuoto o del tennis.

Se poi non hai hobbies o attività di tipo scientifico, sarebbe bene che tu le avessi. Cultura non è solo Musica, Teatro, Prosa e — naturalmente! — Canzoni, ma anche tutto ciò che una volta con un termine perfettamente azzeccato e che sarebbe ora di rispolverare, si chiamava Filosofia Naturale. Personalmente sono convinto che una maggiore diffusione della cultura scientifica produrrebbe frutti preziosi nel Paese. In mancanza (gravissima mancanza!) della Scuola, penso che la libera iniziativa personale, **che c'è**, vada almeno incoraggiata e stimolata, e che questo tipo di cultura vada affrontata con pazienza e umiltà:

"io stimo più il trovare un vero, benché di cosa leggiera, che'l disputar lungamente delle massime quistioni senza conseguir verità nessuna"

Questa frase, che tutti dovrebbero conoscere e meditare ed applicare, porta una firma illustre: Galileo Galilei.

L. 10.000

Cod. 501-A

L'AUTORE

Il Prof. Flavio Waldner si è laureato in Fisica nel 1959 a Padova dove ha lavorato fino al 1967. Dal 1967 al 1969 ha lavorato al Massachussets Institute Of Technology e dal 1970 svolge la sua attività all'Università di Bari dove attualmente è professore straordinario alla cattedra di Laboratorio di Fisica. Ha svolto la sua attività scientifica nel campo delle particelle elementari, partecipando a vari esperimenti in laboratori statunitensi ed europei nell'ambito di collaborazioni internazionali. I suoi interessi scientifici sono attualmente rivolti anche all'astrofisica alla fisica spaziale ed all'elaborazione di dati ed immagini trasmessi via satellite.

GRUPPO
EDITORIALE
JACKSON

Flavio Waldner

IMPARIAMO IL PASCAL

37